RAPPORT TECHNIQUE

TECHNICAL REPORT

CEI
IEC

TR 61131-8

Première édition
First edition
2000-01

Automates programmables –

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

Programmable controllers –

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

Numéro de référence
Reference number
IEC/TR 61131-8:2000

Numéros des publications

Depuis le 1er janvier 1997, les publications de la CEI sont numérotées à partir de 60000.

Publications consolidées

Les versions consolidées de certaines publications de la CEI incorporant les amendements sont disponibles. Par exemple, les numéros d'édition 1.0, 1.1 et 1.2 indiquent respectivement la publication de base, la publication de base incorporant l'amendement 1, et la publication de base incorporant les amendements 1 et 2.

Validité de la présente publication

Le contenu technique des publications de la CEI est constamment revu par la CEI afin qu'il reflète l'état actuel de la technique.

Des renseignements relatifs à la date de reconfirmation de la publication sont disponibles dans le Catalogue de la CEI.

Les renseignements relatifs à des questions à l'étude et des travaux en cours entrepris par le comité technique qui a établi cette publication, ainsi que la liste des publications établies, se trouvent dans les documents ci-dessous:

- **«Site web» de la CEI***
- **Catalogue des publications de la CEI**
  Publié annuellement et mis à jour régulièrement
  (Catalogue en ligne)*
- **Bulletin de la CEI**
  Disponible à la fois au «site web» de la CEI* et comme périodique imprimé

Terminologie, symboles graphiques et littéraux

En ce qui concerne la terminologie générale, le lecteur se reportera à la CEI 60050: *Vocabulaire Electrotechnique International* (VEI).

Pour les symboles graphiques, les symboles littéraux et les signes d'usage général approuvés par la CEI, le lecteur consultera la CEI 60027: *Symboles littéraux à utiliser en électrotechnique,* la CEI 60417: S*ymboles graphiques utilisables sur le matériel. Index, relevé et compilation des feuilles individuelles,* et la CEI 60617: *Symboles graphiques pour schémas.*

* Voir adresse «site web» sur la page de titre.

Numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series.

Consolidated publications

Consolidated versions of some IEC publications including amendments are available. 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.

Validity of this publication

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 the date of the reconfirmation of the publication is available in the IEC catalogue.

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 to be found at the following IEC sources:

- **IEC web site***
- **Catalogue of IEC publications**
  Published yearly with regular updates
  (On-line catalogue)*
- **IEC Bulletin**
  Available both at the IEC web site* and as a printed periodical

Terminology, graphical and letter symbols

For general terminology, readers are referred to IEC 60050: *International Electrotechnical Vocabulary* (IEV).

For graphical symbols, and letter symbols and signs approved by the IEC for general use, readers are referred to publications IEC 60027: *Letter symbols to be used in electrical technology*, IEC 60417: *Graphical symbols for use on equipment. Index, survey and compilation of the single sheets* and IEC 60617: *Graphical symbols for diagrams.*

* See web site address on title page.

RAPPORT TECHNIQUE

TECHNICAL REPORT

CEI
IEC

TR 61131-8

Première édition
First edition
2000-01

Automates programmables –

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

Programmable controllers –

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

International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland
Telefax: +41 22 919 0300 e-mail: inmail@iec.ch IEC web site http://www.iec.ch

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

CODE PRIX
PRICE CODE XC

Pour prix, voir catalogue en vigueur
For price, see current catalogue

SOMMAIRE

Pages

CONTENTS

Articles Pages

Articles Pages

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

AUTOMATES PROGRAMMABLES –

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

AVANT-PROPOS

1) La CEI (Commission Électrotechnique Internationale) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de la CEI). La CEI a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, la CEI, entre autres activités, publie des Normes internationales. Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec la CEI, participent également aux travaux. La CEI collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de la CEI concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux intéressés sont représentés dans chaque comité d'études.
3) Les documents produits se présentent sous la forme de recommandations internationales. Ils sont publiés comme normes, spécifications techniques, rapports techniques ou guides et agréés comme tels par les Comités nationaux.
4) Dans le but d'encourager l'unification internationale, les Comités nationaux de la CEI s'engagent à appliquer de façon transparente, dans toute la mesure possible, les Normes internationales de la CEI dans leurs normes nationales et régionales. Toute divergence entre la norme de la CEI et la norme nationale ou régionale correspondante doit être indiquée en termes clairs dans cette dernière.
5) La CEI n'a fixé aucune procédure concernant le marquage comme indication d'approbation et sa responsabilité n'est pas engagée quand un matériel est déclaré conforme à l'une de ses normes.
6) L'attention est attirée sur le fait que certains des éléments du présent rapport technique peuvent faire l'objet de droits de propriété intellectuelle ou de droits analogues. La CEI ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de propriété et de ne pas avoir signalé leur existence.

La tâche principale des comités d'études de la CEI est l'élaboration des Normes internationales. Toutefois, un comité d'études peut proposer la publication d'un rapport technique lorsqu'il a réuni des données de nature différente de celles qui sont normalement publiées comme Normes internationales, cela pouvant comprendre, par exemple, des informations sur l'état de la technique.

Un rapport technique ne doit pas nécessairement être révisé avant que les données qu'il contient ne soient plus jugées valables ou utiles par le groupe de maintenance.

La CEI 61131-8, qui est un rapport technique, a été établie par le sous-comité 65B: Dispositifs, du comité d'études 65 de la CEI: Mesure et commande dans les processus industriels.

Le texte de ce rapport technique est issu des documents suivants:

Projet d'enquête	Rapport de vote
65B/284/CDV	65B/336/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de ce rapport technique.

Ce document, purement informatif, ne doit pas être considéré comme une Norme internationale.

Cette publication a été rédigée selon les Directives ISO/CEI, Partie 3.

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

PROGRAMMABLE CONTROLLERS –

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

FOREWORD

1) The IEC (International Electrotechnical Commission) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of the IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, the IEC publishes International Standards. Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. The IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of the IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested National Committees.
3) The documents produced have the form of recommendations for international use and are published in the form of standards, technical specifications, technical reports or guides and they are accepted by the National Committees in that sense.
4) In order to promote international unification, IEC National Committees undertake to apply IEC International Standards transparently to the maximum extent possible in their national and regional standards. Any divergence between the IEC Standard and the corresponding national or regional standard shall be clearly indicated in the latter.
5) The IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with one of its standards.
6) Attention is drawn to the possibility that some of the elements of this technical report may be the subject of patent rights. The IEC shall not be held responsible for identifying any or all such patent rights.

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

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

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

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

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

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

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

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

AUTOMATES PROGRAMMABLES –

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

1 Généralités

1.1 Domaine d'application

Le présent rapport technique s'applique à la programmation des systèmes d'automates programmables utilisant les langages de programmation définis dans la CEI 61131-3. Il fournit aussi des lignes directrices pour la mise en oeuvre de ces langages dans les systèmes d'automates programmables et leur environnement de support de programmation (PSE).

Il convient de se rapporter à la CEI 61131-4 pour tous les autres aspects des applications des systèmes d'automates programmables.

NOTE Ni la CEI 61131-3 ni le présent rapport technique ne traitent des problèmes de sécurité des systèmes d'automates programmables ou de leurs logiciels associés. Il convient, pour ces considérations, de consulter la CEI 61508-1 et la CEI 61508-7.

1.2 Documents de références

CEI 60559:1989, *Arithmétique binaire en virgule flottante pour les systèmes à micro processeurs*

CEI 60617-12:1991, *Symboles graphiques pour schémas – Partie 12: Opérateurs logiques binaires*

CEI 60617-13:1993, *Symboles graphiques pour schémas – Partie 13: Opérateurs analogiques*

CEI 60848:1988, *Etablissement des diagrammes fonctionnels pour systèmes de commande*

CEI 61131-1:1992, *Automates programmables – Partie 1: Informations générales*

CEI 61131-2:1992, *Automates programmables – Partie 2: Spécifications et essais des équipements*

CEI 61131-3:1993, *Automates programmables – Partie 3: Langages de programmation*

CEI 61131-4:1995, *Automates programmables – Partie 4: Guide pour l'utilisateur*

CEI 61131-5:1998, *Automates programmables – Partie 5: Communications*

CEI 61508-1:1998, *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 1: prescriptions générales*

PROGRAMMABLE CONTROLLERS –

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

1 General

1.1 Scope

This technical report applies to the programming of programmable controller systems using the programming languages defined in IEC 61131-3. It also provides guidelines for the implementation of these languages in programmable controller systems and their programming support environments (PSEs).

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

NOTE Neither IEC 61131-3 nor this technical report explicitly addresses safety issues of programmable controller systems or their associated software. IEC 61508-1 and IEC 61508-7 should be consulted for such considerations.

1.2 Reference documents

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

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

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

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

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

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

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

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

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

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

CEI 61508-2, — *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 2: Prescriptions pour les systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité* [1)]

CEI 61508-3:1998, *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 3: Prescriptions concernant les logiciels*

CEI 61508-4:1998, *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 4: Définitions et abréviations*

CEI 61508-5:1998, *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 5: Exemples de méthodes de détermination concernant les logiciels*

CEI 61508-6, — *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 6: Lignes directrices pour l'application de la CEI 1508-3* [1)]

CEI 61508-7, — *Sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables relatifs à la sécurité – Partie 7; Présentation de techniques de mesures* [1)]

ISO/CEI 646:1991, *Technologies de l'information – Jeu ISO de caractères codés à 7 éléments pour l'échange d'informations*

ISO/CEI 9506-5:1999, *Système d'automatisation industrielle, Spécification de messagerie industrielle – Partie 5: Norme d'accompagnement pour les automates programmables* [1)]

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

NOTE Les dispositions techniques et normatives de la CEI 60559 et de l'ANSI/IEEE Std 754 sont équivalentes.

1.3 Généralités

L'audience prévue pour le présent rapport technique comprend:

– les utilisateurs, tels qu'ils sont définis dans la CEI 61131-3, des systèmes d'automates programmables, qui doivent programmer, configurer, installer et maintenir des automates programmables faisant partie des processus industriels de mesurage et de commande;

– les implémenteurs de langages de programmation pour les systèmes d'automates programmables, tels qu'ils sont définis dans la CEI 61131-3. Ce groupe peut inclure des vendeurs de *logiciel* et de *matériel* pour la préparation et la maintenance de *programmes* pour ces systèmes, ainsi que les vendeurs des systèmes d'automates programmables eux-mêmes.

L'article 2 du présent rapport technique fournit une introduction générale à la CEI 61131-3 sur les systèmes d'automates programmables, alors que l'article 3 fournit des informations supplémentaires concernant l'application de certains éléments de langages de programmation spécifiés par la CEI 61131-3. L'article 4 fournit des informations relatives à la mise en oeuvre prévue de certains éléments de ces langages de programmation, alors que l'article 5 donne des informations générales sur les besoins en matériel et en logiciel pour le développement et la maintenance de programmes. Il est donc prévu que les utilisateurs d'automates programmables trouveront très utiles les articles 2 et 3 du présent rapport technique, alors que les implémenteurs de langages de programmation trouveront plus d'utilité aux articles 4 et 5, en se référant, si nécessaire, aux informations de base des articles 2 et 3.

---

1) A publier.

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

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

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

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

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

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

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

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

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

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

1.3 Overview

The intended audience for this technical report consists of:

– *users* of *programmable controller systems* as defined in IEC 61131-3, who must program, configure, install and maintain programmable controllers as part of industrial process measurement and control systems;

– *implementors* of *programming languages*, as defined in IEC 61131-3, for programmable controller systems. This may include vendors of *software* and *hardware* for the preparation and maintenance of *programs* for these systems, as well as vendors of the programmable controller systems themselves.

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

1) To be published.

2 Introduction à la CEI 61131-3

2.1 Considérations générales

Les possibilités limitées des composants de matériels coûteux ont, dans le passé, imposé de sévères contraintes au processus d'étude des systèmes de commande industriels, de mesurage et d'automatisation. L'étude et la mise en oeuvre du logiciel étaient étroitement dépendantes du matériel choisi. Les spécialistes nécessaires étaient hautement qualifiés, à la fois pour la résolution des problèmes de processus d'automatisation et pour la manipulation d'ensembles de programmation, compliqués et dépendants souvent du matériel.

Avec les innovations rapides dans la micro-électronique et les technologies qui s'y rattachent, le rapport prix/performance des matériels a considérablement décru. Un petit automate programmable peut maintenant coûter bien moins cher que ne coûte sa programmation.

Le coût rapidement décroissant du matériel a entraîné une tendance à remplacer de grands ordinateurs centraux de processus ou d'autres automates autonomes de taille relativement grande par des systèmes distribués spatialement et fonctionnellement.

Comme le montre la figure 1, le sous-système de communications, qui fournit les mécanismes d'échange d'informations entre les dispositifs d'automatisation, constitue le fondement de ces systèmes. Il y a, connectés à ce sous-système, des dispositifs tels que des automates programmables qui fournissent la puissance de traitement distribué du système. Chaque dispositif, contrôlé par son propre logiciel, exécute une sous-tâche qui lui est affectée afin de réaliser la totalité des fonctionnalités du système. On choisit la taille et la puissance de chaque dispositif de façon à satisfaire les demandes de cette sous-tâche particulière.

Historiquement, comme le montre la figure 2, on a utilisé les automates programmables pour des applications autonomes. Les utilisateurs de ces applications bénéficient également de l'évolution esquissée plus haut. A cause des faibles coûts actuels, de nombreuses nouvelles tâches d'automatisation, relativement petites, peuvent être résolues avec souplesse et profit par des automates programmables.

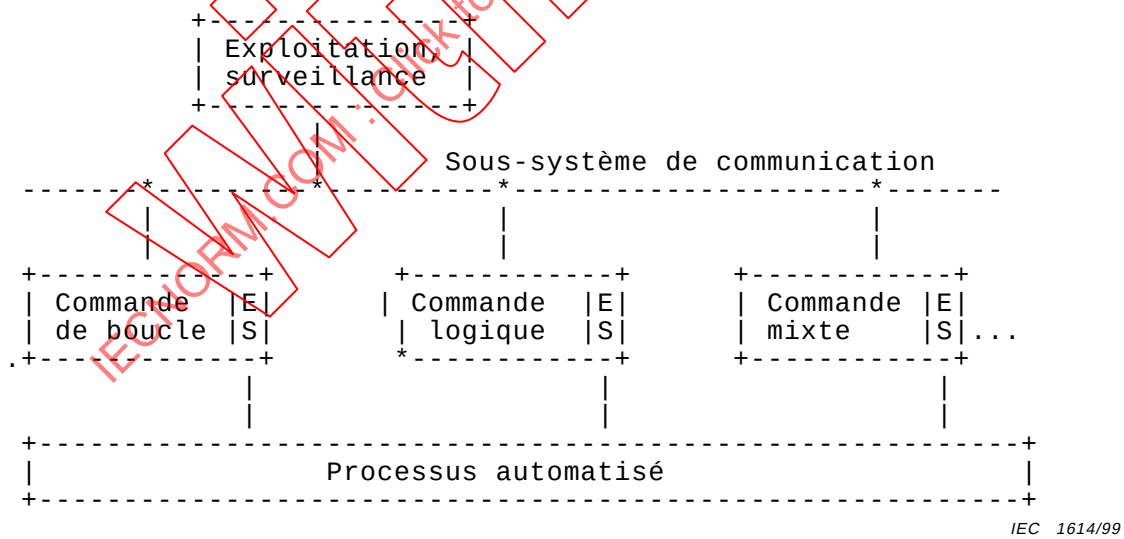


Figure 1 – Application distribuée

2 Introduction to IEC 61131-3

2.1 General considerations

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

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

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

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

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

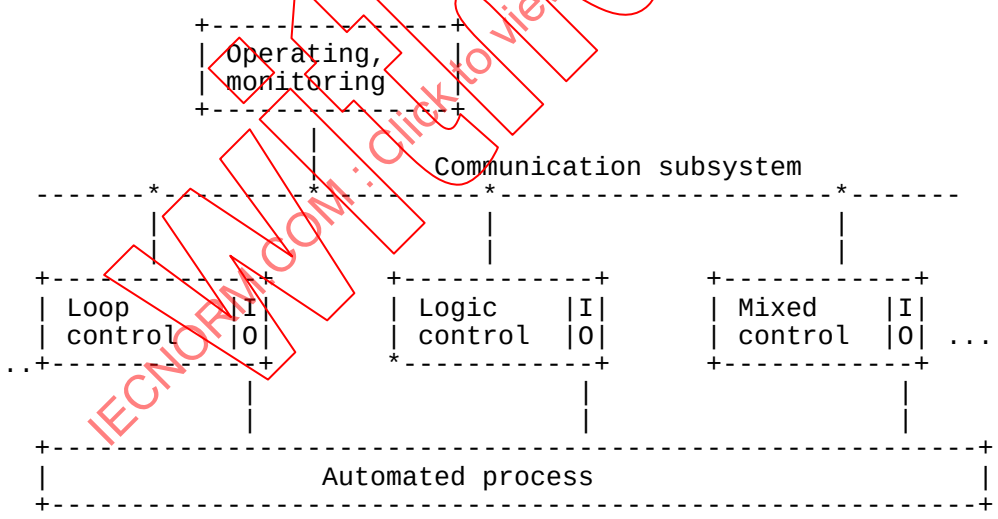


Figure 1 – Distributed application

IEC 1615/99

Figure 2 – Applications autonomes

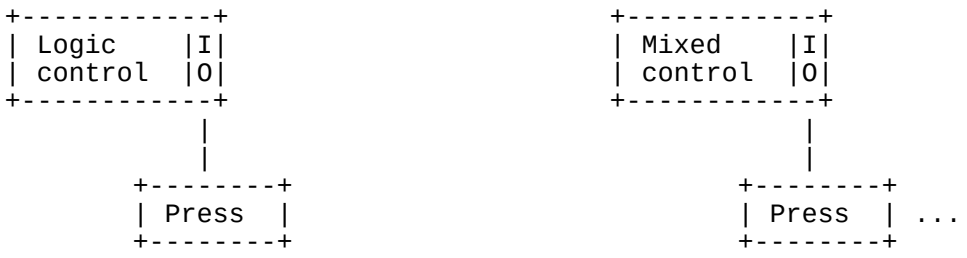
S'ajoutant au faible coût du matériel, la simplicité de leurs principes de programmation et de leurs système d'exploitation, qui sont facilement compris et appliqués par le personnel de base impliqué dans la programmation, l'exploitation et la maintenance, fait également progresser l'usage intensif des automates programmables pour la résolution des problèmes des tâches d'automatisation.

Les automates programmables utilisent typiquement, comme le montre la figure 3, le principe d'une exécution cyclique des programmes. Ce principe est bien connu et appliqué pour l'exploitation des systèmes de traitement des signaux numériques pour simuler le fonctionnement continu d'un système analogique ou électromécanique. Les valeurs du processus sont lues dans les dispositifs et écrites par le processus, comme des échantillons discrets prélevés de façon aléatoire ou à des intervalles de temps fixes, en fonction de la tâche de surveillance qui doit être accomplie.

Ces principes d'exploitation permettent l'élaboration de programmes pour les automates programmables qui utilisent des éléments très proches des principes des logiques câblées ou des circuits de commande continus utilisés auparavant dans ce cas.

Les principes d'exploitation des automates programmables permettent ainsi des langages de programmation spécifiques aux applications. Associés à l'interface homme/système appropriée, ces langages permettent à l'ingénieur chargé de la surveillance de se consacrer à la solution des tâches de surveillance sans formation intensive en génie logiciel. Les spécifications technologiques de l'ingénierie de surveillance peuvent directement correspondre aux éléments du langage.

Un des avantages particuliers des langages de programmation graphiques est qu'ils proposent une représentation qui ne sert pas uniquement à la saisie et à la documentation des programmes, mais aussi aux essais en ligne et au diagnostic. Les environnements de support de programmation (PSE) des automates programmables sont donc capables de fournir une représentation et une documentation graphique qui sont déjà connues de l'ingénieur d'application et du personnel de base.



IEC 1615/99

Figure 2 – Stand-alone applications

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

Programmable controllers typically employ the principle of cyclic program execution illustrated in figure 3. This principle is well known and applied in the operation of digital signal processing systems to simulate the operation of continuously operating analogue or electromechanical systems. Process values are read into the device and written out to the process as discrete samples at random or equidistant points of time, depending on the control task that has to be fulfilled.

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

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

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

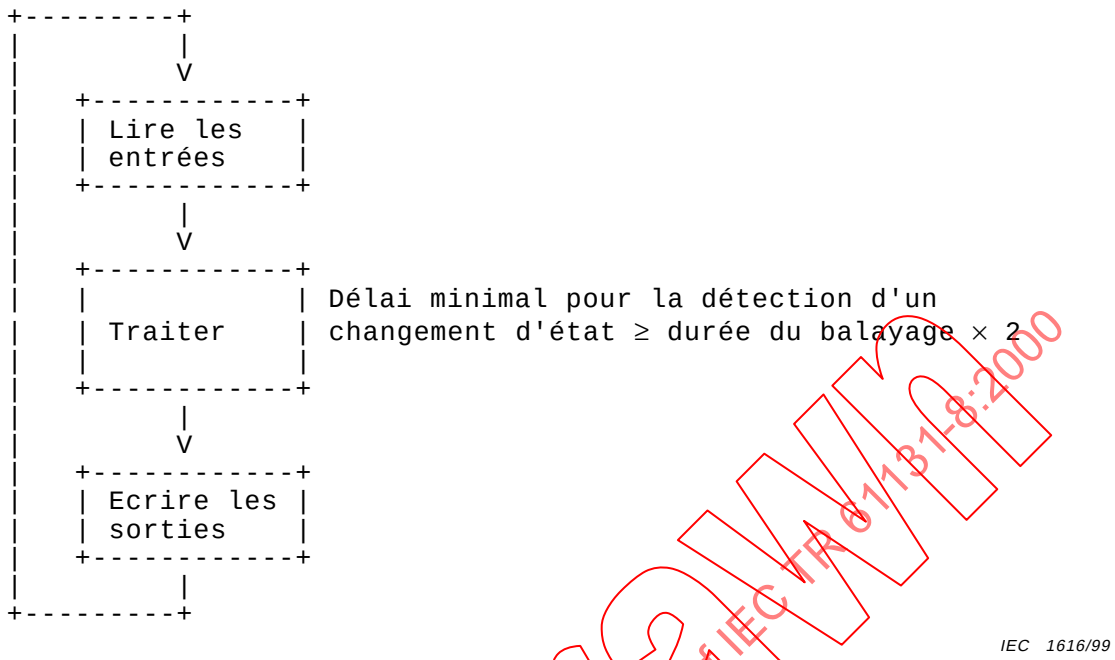


Figure 3 – Balayage d'un programme cyclique

2.2 Limitations historiques

On demande souvent à ceux qui conçoivent les systèmes d'automatisation d'utiliser des automates programmables fournis par différents fabricants dans différents système ou même dans le même système. Les automates programmables de constructeurs différents peuvent, cependant, ne pas avoir grand chose de commun sur le plan matériel. Historiquement, il en a résulté des différences significatives dans les éléments et les méthodes de programmation des logiciels, ce qui a conduit au développement d'outils de programmation et de débogage propres à chaque fabricant, qui contiennent souvent des logiciels spécialisés pour la programmation, les essais et la maintenance d'une «famille» particulière d'automates programmables.

Passer d'une famille d'automates à une autre exige souvent, de la part des concepteurs, la lecture de gros manuels relatifs au matériel et au logiciel de la nouvelle famille. Il faut souvent examiner plusieurs fois les manuels afin d'en comprendre la signification exacte et pour pouvoir se servir de la nouvelle famille d'automates de façon correcte. A cause de la concentration sur la tâche ennuyeuse nécessaire à la lecture et à la compréhension du nouveau matériel spécifique à un fournisseur, peu de gens le font. Pour cette raison, beaucoup de gens considèrent l'étude et la programmation de ces automates comme une espèce de magie noire à éviter. Le savoir-faire pour l'utilisation de ces systèmes est en conséquence effectivement concentré entre les mains d'un ou de quelques spécialistes, et ne peut pas être réellement transmis à ceux qui ont la responsabilité de l'exploitation, de la maintenance et de l'amélioration du système.

L'objectif majeur de la CEI 61131-3 est de supprimer les obstacles à la compréhension et à l'utilisation des automates programmables. En conséquence, la CEI 61131-3 introduit de nombreuses facilités pour renforcer les avantages des automates programmables décrits en 2.1, même quand les automates concernés proviennent de différents fournisseurs. On espère que l'extension des domaines d'application des automates programmables qui en résultera compensera les coûts encourus par les fournisseurs pour rendre les systèmes conformes à la norme.

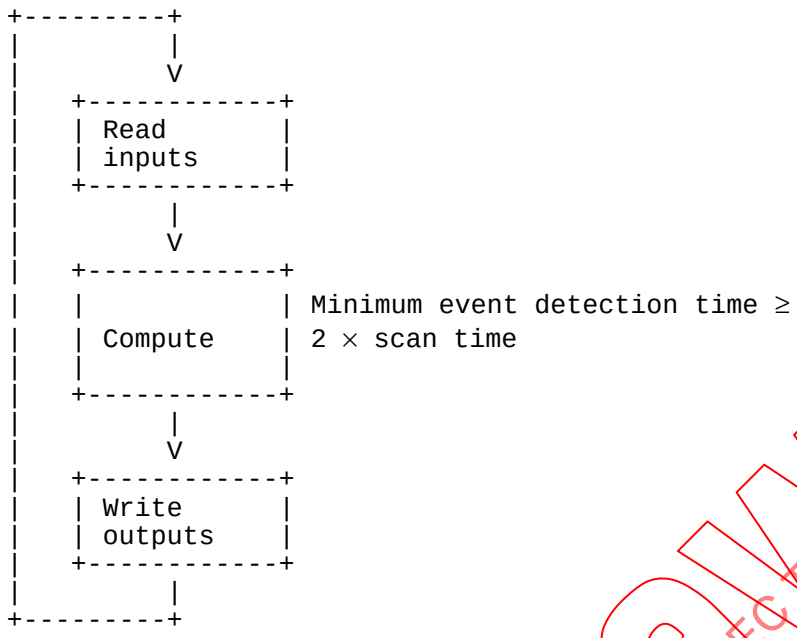


Figure 3 – Scanning a cyclic program

2.2 Historical limitations

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

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

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

2.3 Nouveautés de la CEI 61131-3

Du point de vue de l'ingénieur d'application et de celui du configurateur du système, on peut résumer les nouveautés les plus importantes de la CEI 61131-3 de la façon suivante.

1) l'élaboration bien structurée de programmes ascendants ou descendants est facilitée par des structures de langage pour la définition d'objets fonctionnels typés (program organisation unit, unité d'organisation du programme), tels que fonctions, blocs fonctionnels ou programmes.
2) l'utilisation de données clairement typées n'est pas seulement encouragée, mais fondamentalement exigée, éliminant une importante cause d'erreurs de programmation.
3) un jeu suffisant de dispositifs pour le suivi de l'exécution des unités d'organisation de programmes est inclus: ces dispositifs, associés aux blocs d'étapes, de transition et d'actions, offrent un excellent moyen de représenter, sous une forme concise, des solutions de suivi séquentielles compliquées.
4) les fonctionnalités nécessaires à la conception de la communication entre programmes d'application sont prévues. Des fonctions de communication identiques peuvent être utilisées entre deux programmes, qu'ils soient ou non sur le même appareil, ce qui facilite la réutilisation du logiciel dans différents environnements.
5) possibilité pour le concepteur de choisir deux langages graphiques et deux langages littéraux, en fonction des besoins de l'application. Ces langages, avec un jeu de «common elements» (éléments communs) graphiques ou littéraux, fournissent un support logiciel aux méthodologies de conception basées sur des modèles bien compris:
   a) le langage graphique à contacts (LD = *ladder diagram*) modélise des réseaux d'éléments électromécaniques tels que relais à contacts et à bobines, temporisateurs, compteurs, etc., fonctionnant simultanément;
   b) le langage graphique à blocs fonctionnels (FBD = *function block diagram*) modélise un réseau d'éléments électroniques tels qu'additionneurs, multiplicateurs, registres à décalages, portes, etc., fonctionnant simultanément;
   c) le langage littéral structuré (ST = *structured text*) modélise les informations typiques des tâches de traitement telles que des algorithmes numériques utilisant des structures que l'on trouve dans les langages de haut niveau à usage général, comme Pascal;
   d) le langage liste d'instructions (IL = *instruction list*) modélise un bas niveau de programmation des systèmes de surveillance en langage assembleur;
   e) un jeu de common elements graphiques et littéraux fournit des règles pour la définition des valeurs et des variables, des dispositifs pour la configuration des logiciels et les déclarations d'objets. Les common elements contiennent des éléments graphiques et littéraux pour la construction de:
   f) sequencial function charts (SFC = diagramme fonctionnel en séquence), qui modélisent des dispositifs et des algorithmes de commande séquentiels basés sur le temps ou les changements d'états.
6) La souplesse dans le choix des langages convenant à la programmation de fonctionnalités spécifiques aux applications augmentera la réutilisation de solutions logicielles à des problèmes de surveillance de processus.
7) Chaque spécialiste d'application au sein d'une équipe projet peut utiliser le style et le langage qui convient à l'application spécifique considérée, avec la garantie que les résultats du travail de tous les spécialistes seront facilement intégrés tous ensemble.

En résumé, l'objectif principal de la CEI 61131-3 est d'introduire tous les concepts et les structures de langages normalisés nécessaires à la solution des problèmes technologiques de chaque application, et de fournir les principes de développement d'éléments logiciels indépendants du constructeur. Cela rend plus facile la réutilisation des conceptions de logiciels de surveillance de différents types d'automates, même s'il faut encore faire quelque efforts pour passer un programme de surveillance d'un automate à un autre.

2.3 New features in IEC 61131-3

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

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

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

2.4 Considérations sur l'ingénierie des logiciels

2.4.1 Mesurage de la qualité des logiciels

La qualité des systèmes pilotés par des logiciels se mesure à l'aide d'un certain nombre d'attributs [1][1)]. Les plus importantes parmi les mesures de la qualité des systèmes de processus industriels de surveillance et d'automatisation sont:

1) *capacité:* limite jusqu'à laquelle le système peut assurer les fonctions pour lesquelles il est prévu. Le mesurage de la capacité comprend:
   a) *temps de réponse:* temps qu'il faut au système pour produire les réponses appropriées à une combinaison spécifiée de changements d'états externes;
   b) *capacité de traitement:* limite jusqu'à laquelle le système peut respecter les délais planifiés avec un ensemble spécifié de conditions;
   c) *capacité de stockage:* limite jusqu'à laquelle le système peut garder en mémoire tous les programmes et les données nécessaires avec un ensemble spécifié de conditions.
2) *disponibilité:* proportion de temps que le système est capable de consacrer à l'exécution des fonctions pour lesquelles il est prévu. La disponibilité varie en fonction de facteurs tels que:
   a) *fiabilité:* capacité du système à continuer à remplir les fonctions pour lesquelles il est prévu, pendant une période de temps et avec un ensemble de conditions spécifiés. Une autre mesure de fiabilité est le temps moyen entre pannes (MTBF = mean time between failure);
   b) *maintenabilité:* facilité avec laquelle on peut remettre le système dans son état de marche intégral après une ou plusieurs pannes dans un domaine défini. Une autre mesure de la maintenabilité est le temps moyen de remise en état (MTTR = mean time to repair); la maintenabilité est souvent définie par l'expression MTBF/(MTBF+MTTR);
   c) *intégrité:* degré jusqu'où le système peut continuer à remplir les fonctions pour lesquelles il est prévu malgré un nombre défini d'atteintes, y compris les actions involontaires des utilisateurs, les actions volontairement nuisibles, ainsi que les applications potentiellement dangereuses et les changements d'états du système.
3) *facilité d'utilisation:* facilité avec laquelle un ensemble défini d'utilisateurs peut acquérir et exercer la possibilité de coopérer avec le système pour remplir les fonctions pour lesquelles ce dernier est prévu. La capacité d'utilisation dépend des facteurs suivants:
   a) *niveau exigé:* niveau de formation formel ou informel nécessaire avant qu'un utilisateur puisse apprendre à coopérer avec le système (par exemple niveau de formation, expérience de l'utilisation des systèmes d'exploitation, des systèmes de fenêtrage, etc.).
   b) *besoins en formation:* formation nécessaire pour qu'un utilisateur d'un niveau donné puisse coopérer avec le système pour remplir un ensemble défini de fonctions du système;
   c) *productivité de l'utilisateur:* nombre d'opérations liées au système effectuées par unité de temps par un utilisateur d'un niveau donné de formation et d'expérience;
   d) *convivialité:* degré jusqu'où un utilisateur préfère l'utilisation du logiciel du système pour effectuer les fonctions prévues du système à l'utilisation d'activités alternatives qui pourraient ne pas être liées au système.
4) *adaptabilité:* facilité avec laquelle la fonctionnalité du système peut être changée de diverses façons comme:
   a) *faculté d'amélioration:* facilité avec laquelle la *capacité*, la *disponibilité* et/ou les *facilités d'utilisation* du système existant peuvent être augmentées sans changement fondamental des fonctionnalités du système;
   b) *extensibilité:* facilité avec laquelle on peut ajouter des fonctionnalités au système;

---

1) Les chiffres entre crochets se rapportent à la bibliographie

2.4 Software engineering considerations

2.4.1 Software quality measures

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

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

[1)] Figures in square brackets refer to the bibliography.

c) *portabilité:* facilité avec laquelle les fonctionnalités d'un système peuvent être transférées d'un système à un autre;

d) *réutilisabilité*: facilité avec laquelle on peut utiliser les possibilités fonctionnelles d'un élément de logiciel existant pour augmenter les capacités d'un système, qu'il soit nouveau ou non.

2.4.2 Application des principes d'ingénierie des logiciels

On a utilisé différents principes d'ingénierie des logiciels lors de l'élaboration de la CEI 61131-3 afin de promouvoir une meilleure qualité du logiciel. Quelques-uns des plus importants de ces principes, leur contribution à la qualité du logiciel, ainsi que leur intégration dans la CEI 61131-3 sont discutés ci-dessous.

2.4.2.1 Encapsulage et masquage

L'encapsulage est le «conditionnement» de données et/ou de procédures reliées fonctionnellement dans une seule entité logicielle. L'encapsulage contribue à la fiabilité, à la maintenabilité, à la facilité d'utilisation et à l'adaptabilité des logiciels.

On associe à l'encapsulage la notion de masquage de procédures et de données, pour laquelle la seule connaissance que l'utilisateur a d'une entité logicielle est son interface externe ainsi que ses fonctionnalités spécifiées. Les détails de la structure interne des données et de la mise en oeuvre des procédures sont cachés volontairement. Le masquage contribue à la maintenabilité, l'intégrité, la facilité d'utilisation, la portabilité et la réutilisabilité du logiciel.

Les éléments de la CEI 61131-3 qui traitent de l'encapsulage et du masquage, ainsi que les principaux paragraphes s'y rapportant dans la CEI 61131-3 sont énumérés au tableau 1.

Tableau 1 – Eléments de la CEI 61131-3 traitant de l'encapsulage et du masquage

Elément (paragraphe)	Encapsulage		Masquage	
	Données	Procédures	Données	Procédures
Structure (2.3.3)	Oui	Non	Non	Non
Fonction (2.5.1)	Non	Oui	Oui	Oui
Bloc fonctionnel (2.5.2)	Oui	Oui	Oui	Oui
Programme (2.5.3)	Oui	Oui	Oui	Oui
Action (2.6.4)	Non	Oui	Non	Non
Chemin d'accès (2.7.1)	Oui	Non	Oui	Non

2.4.2.2 Représentation explicite d'état

Les éléments du SFC (sequential function chart = diagramme fonctionnel en séquence) définis en 2.6 de la CEI 61131-3 permettent à tout instant de déterminer l'état du système de surveillance par un ensemble d'étapes et d'actions actives. Sans cette représentation, il faut déduire l'état du système à partir de données telles que les entrées et les sorties du système ainsi que de certaines variables d'état (booléennes). L'utilisation des éléments SFC contribue donc à la maintenabilité, à la facilité d'utilisation et à la portabilité du logiciel. Les temps de réponse et les capacités de traitements sont en outre améliorés en ne traitant que la partie des algorithmes qui dépend de l'état présent.

2.4.2.3 Correspondance avec le domaine d'application

On a déjà remarqué, en 2.1 et 2.3 du présent rapport technique, la correspondance directe qui existe entre les éléments de la CEI 61131-3 et des concepts bien connus pour le mesurage, l'automatisation et la surveillance des processus industriels. Cette caractéristique contribue à la maintenabilité, à la facilité d'utilisation et à l'adaptabilité du logiciel.

c) *portability:* the ease with which the system functionality can be moved from one system to another;

d) *reusability:* the ease with which the functional capabilities of an existing software element can be used to add capability to a new or existing system.

2.4.2 Application of software engineering principles

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

2.4.2.1 Encapsulation and hiding

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

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

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

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

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

2.4.2.2 Explicit representation of state

The sequential function chart (SFC) elements defined in 2.6 of IEC 61131-3 enable the state of the control system to be determined at any point in time as the set of active steps and actions. Without this representation, the state of the system must be inferred from data such as system inputs, outputs and some set of state (Boolean) variables. The use of SFC elements thus contributes to software maintainability, usability and portability. Furthermore, system responsiveness and processing capacity are enhanced by performing only those portions of the software algorithm relevant to the current state.

2.4.2.3 Mapping to the application domain

The direct mapping of the elements of IEC 61131-3 to well-understood concepts in industrial process measurement, automation and control has already been noted in 2.1 and 2.3. This characteristic contributes to software maintainability, usability and adaptability.

2.4.2.4 Correspondance entre la conception et la mise en oeuvre

La CEI 61131-3 préconise un style de réalisation de système connu sous le nom de «conception descendante» (ou «conception par décomposition fonctionnelle») suivi d'une «mise en oeuvre ascendante» (ou «mise en oeuvre par assemblage fonctionnel»). Cela contribue à la fiabilité, la maintenabilité, la facilité d'utilisation et à l'adaptabilité du logiciel.

Ce style de conception et de mise en oeuvre du système se caractérise par la séquence d'étapes ci-dessous.

1) Les fonctionnalités désirées du système ainsi que les interfaces externes sont spécifiées. Par exemple, les fonctionnalités de base d'une cellule d'usinage peuvent devoir accepter une pièce brute d'un système de manipulation de matériels, produire une pièce finie à partir de l'ébauche brute, mesurer la pièce finie, de la rendre au système de manipulation de matériels, et rendre compte des résultats de l'opération au système d'informations de la fabrication. Les interfaces externes de cette cellule incluent les interfaces avec le système de manipulation de matériels et avec le système d'information. On peut, à ce stade, utiliser les éléments de configuration décrits en 2.7 de la CEI 61131-3.

2) On peut procéder à une première subdivision de la conception du système en allouant les fonctionnalités exigées à un ou plusieurs éléments, typiquement des programmes (voir 2.5.3 de la CEI 61131-3). Les interfaces entre les programmes et entre les programmes et les interfaces externes du système sont définies ainsi que la fonction affectée à chaque programme. Cette subdivision suit souvent la partition physique du système; par exemple, dans la cellule d'usinage décrite plus haut, on pourrait définir des programmes séparés pour la station d'usinage, la station de mesurage et, le cas échéant, pour le robot de manipulation des matériels de la cellule.

3) Chacun des éléments définis lors de l'étape précédente est ensuite subdivisé en unités plus élémentaires. Si la fonction de l'élément est essentiellement séquentielle, la première phase de la subdivision pourrait être la formulation d'un SFC (diagramme fonctionnel en séquence = sequential function chart) (voir 2.6 de la CEI 61131-3) exprimant la séquence des opérations à effectuer et les conditions pour répéter le cycle des opérations. Chaque action (voir 2.6.4 de la CEI 61131-3) du SFC est alors encore subdivisée, typiquement en *blocs fonctionnels* (voir 2.5.2 de la CEI 61131-3), c'est-à-dire en FBD (*diagramme de blocs fonctionnels* = function block diagram) (voir 4.3 de la CEI 61131-3). Le programme principal pour la station d'usinage dans la cellule décrite plus haut pourrait, par exemple, être un SFC décrivant la séquence des opérations d'usinage à effectuer, alors que les actions pourraient contenir des blocs fonctionnels effectuant les fonctions de surveillance des mouvements nécessaires.

4) Ce processus de subdivision fonctionnelle est réalisé récursivement jusqu'à ce que toutes les fonctionnalités puissent être identifiées comme appartenant à des éléments existant dans une bibliothèque (voir 1.4.3 de la CEI 61131-3), ou qu'elles puissent être exprimées de façon algorithmique à l'aide d'un des langages littéraux ou graphiques de la CEI 61131-3, c'est-à-dire le langage *littéral structuré (ST)* (voir 3.2 de la CEI 61131-3), *Liste d'instructions (IL)* (voir 3.1 de la CEI 61131-3), *à contact (LD)* (voir 4.2 de la CEI 61131-3), ou *à blocs fonctionnels (FBD)* (voir 4.3 de la CEI 61131-3).

5) Le système est ensuite mis en oeuvre par un assemblage fonctionnel «ascendant», c'est-à-dire en compilant et en ajoutant à la bibliothèque les éléments nouvellement définis dans l'ordre inverse de l'ordre dans lequel ils ont été définis dans les étapes précédentes. Avec un peu d'attention lors de la conception pour la réutilisation, beaucoup de ces nouveaux éléments de la bibliothèque pourront servir lors de la conception de systèmes futurs.

6) Finalement, l'affectation des programmes aux ressources, et des ressources aux configurations est terminée, les tâches d'exécution des programmes définies et les chemins d'accès établis pour la communication avec le système d'informations à l'aide des éléments de configuration définis en 2.7 de la CEI 61131-3.

2.4.2.4 Mapping of design to implementation

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

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

1) The system’s desired functionality and external interface are specified. For instance, the basic functionality of a machining cell might be to accept a rough part from a material handling system, procure a finished part from the rough blank, measure the finished part, pass it back to the material handling system and report the results of the operation to a manufacturing information system. External interfaces in this cell would include the material handling and information system interfaces. The *configuration elements* described in 2.7 of IEC 61131-3 can be used in this step.
2) A first decomposition of the system design is made by allocating the required functionality into one or more elements, typically *programs* (see 2.5.3 of IEC 61131-3). The interfaces among the programs and between the programs and the system’s external interfaces, are defined and the functionality allocated to each program is defined. Such a decomposition will often follow the physical portioning of the system; for instance, in the machining cell described above, separate programs might be defined for the machining station, the measuring station and for the cell’s material handling robot, if any.
3) Each element defined in the preceding step is further decomposed into more basic functional units. If the functionality of the element is essentially sequential, the first step in the decomposition may be the formulation of a *sequential function chart* (SFC) (see 2.6 of IEC 61131-3) expressing the sequence of operations to be performed and the conditions for repeating the cycle of operations. Each *action* (see 2.6.4 of IEC 61131-3) of the SFC is then further decomposed, typically into interconnected *function blocks* (see 2.5.2 of IEC 61131-3), i.e. a *function block diagram* (FBD) (see 4.3 of IEC 61131-3). For instance, the main program for the machining station in the cell described above may be a SFC describing the sequence of machining operations to be performed, whilst the actions might contain function blocks performing the required motion control functions.
4) This functional decomposition process is performed recursively until all functionality can be identified as belonging to existing library elements (see 1.4.3 of IEC 61131-3) or can be expressed algorithmically in one of the textual or graphic languages in IEC 61131-3, i.e. *structured text* (ST) (see 3.2 of IEC 61131-3), *instruction list* (IL) (see 3.1 of IEC 61131-3), *ladder diagram* (LD) (see 4.2 of IEC 61131-3) or *function block diagram* (FBD) (see 4.3 of IEC 61131-3).
5) The system is then implemented by “bottom up” functional composition, i.e. by compiling and adding to the library the newly defined elements in the reverse order in which they have been defined in the preceding steps. With some attention to design for reusability, many of the new library elements may be usable in future system designs.
6) Finally, the allocation of *programs* to *resources* and *resources* to *configurations* is completed, program execution tasks are set up and *access paths* are established for communication with information systems, using the configuration elements defined in 2.7 of IEC 61131-3.

2.4.2.5 Programmation structurée

Les techniques de programmation structurée contribuent, c'est bien connu, à la fiabilité, à la maintenabilité et à l'adaptabilité du logiciel. Le langage littéral structuré (ST), défini en 3.3 de la CEI 61131-3, fournit un ensemble complet de structures pour supporter ce style de programmation, tout en restant compatible avec les autres langages graphiques et littéraux ainsi qu'avec les éléments de la CEI 61131-3.

2.4.2.6 Réutilisation du logiciel

Le modèle de programmation décrit en 1.4.3 de la CEI 61131-3 et illustré à la figure 3 de la CEI 61131-3, supporte la réutilisation des éléments de logiciels, ce qui peut être développé par l'utilisateur dans le cadre du processus ascendant de mise en oeuvre décrit en 2.4.2.4 ou peut être fourni sous forme de bibliothèques par les fournisseurs de logiciel. Cette méthode de construction d'un système peut ne pas être familière aux utilisateurs qui ont, dans le passé, développé des applications pour des systèmes d'automatisation sous la forme d'un unique grand contact. Une certaine formation peut donc s'avérer nécessaire pour réaliser les importants gains potentiels de qualité du logiciel et de productivité procurés par la réutilisation du logiciel présentée dans la CEI 61131-3.

Comme on le décrit en 1.4.3 et à l'article B.0 de la CEI 61131-3, les éléments de logiciel que l'on peut mettre dans une bibliothèque pour être réutilisés comprennent, dans un ordre de complexité et de fonctionnalité croissant:

– des types de données;
– des fonctions;
– des blocs fonctionnels;
– des programmes;
– des configurations.

2.4.3 Portabilité

Comme on le note en 2.4.1 ci-dessus, la *portabilité* est définie comme étant la facilité avec laquelle on transporte une fonctionnalité de système d'un système à un autre. Ce concept peut être considéré de deux points de vue:

– *la portabilité entre langages*, c'est-à-dire la facilité avec laquelle on convertit une spécification de type d'unité d'organisation de programme d'un langage à un autre, ou
– *la portabilité entre systèmes*, c'est-à-dire la facilité avec laquelle on convertit une spécification de type d'unité d'organisation de programme d'un environnement d'aide à la programmation (PSE) vers un autre.

2.4.3.1 Portabilité entre langages

Comme on le note en 2.4.2.1, les facilités d'*encapsulage* et de *masquage* de la CEI 61131-3 fournissent un degré très élevé de *réutilisabilité* des fonctions, des blocs fonctionnels et des types de données dans tous les langages définis. Cependant, comme on le note au point 5 de 2.3, chaque langage de le CEI 61131-3 est, jusqu'à un certain point, spécialisé pour un modèle particulier du domaine de problèmes. Cela limite la facilité avec laquelle un algorithme écrit dans un des langages CEI peut être traduit dans un autre langage de programmation. Par exemple:

– les structures de sélections et les itérations du langage ST sont difficiles à traduire efficacement dans les langages FBD ou LD;
– les entrées EN et les sorties ENO des fonctions peuvent être représentées directement dans les langages LD et FDB, mais pas dans les langages ST ou IL;

2.4.2.5 Structured programming

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

2.4.2.6 Software reuse

The programming model described in 1.4.3 of IEC 61131-3 and shown in figure 3 of IEC 61131-3 strongly supports the reuse of software elements. These may be developed by the user in the "bottom-up" implementation process described in 2.4.2.4 above, or may be supplied as "libraries" by software vendors. This method of system construction may be unfamiliar to users who have previously developed automation system applications as a single, large ladder diagram. Hence, some training may be required in order to realize the large potential gains in software quality and productivity presented by the new approach to software reuse presented in IEC 61131-3.

As described in 1.4.3 and clause B.0 of IEC 61131-3, the software elements that may be placed in libraries for reuse include, in order of increasing complexity and functionality:

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

2.4.3 Portability

As noted in 2.4.1 above, *portability* is defined as the ease with which system functionality can be moved from one system to another. This may be considered from the point of view of:

– *inter-language portability*, i.e. the ease of converting a program organization unit type specification from one language to another, or
– *inter-system portability*, i.e. the ease of converting a program organization unit type specification from one programming support environment (PSE) to another.

2.4.3.1 Inter-language portability

As noted in 2.4.2.1, the *encapsulation* and *hiding* facilities of IEC 61131-3 provide a high degree of *reusability* of functions, function blocks and data types among all the defined languages. However, as noted in item 5) of 2.3, each of the IEC 61131-3 languages is specialized to some extent for a particular model of the problem domain. This limits the ease with which an algorithm written in one of the IEC languages can be translated into another programming language. For instance:

– the selection and iteration constructs of the ST language are difficult to translate efficiently into FBD or LD;
– the EN input and ENO output of functions can be represented directly in the LD and FBD languages, but not in the ST or IL language;

– les expressions littérales ne peuvent être utilisées que dans le langage ST, pas dans les langages LD ou FBD.

L'utilisateur peut donc choisir le langage le plus approprié au type d'algorithme à développer au sein d'une fonction ou d'un bloc fonctionnel.

2.4.3.2 Portabilité entre systèmes

La CEI 61131-3 ne définit ni n'exige de définition de type de format commune pour l'échange d'unités d'organisation de programme (POU). En conséquence, les POU conformes, telles que définies en 1.5.2 de la CEI 61131-3, ne seront facilement portables que si elles sont écrites dans un langage littéral (ST ou IL). Les POU conformes ne seront portables que si les dispositifs du système cible sont égaux à ceux du système origine ou en constituent un sur-ensemble.

NOTE Un format d'échange commun est à l'étude en tant qu'extension de la CEI 61131-3.

3 Lignes directrices pour les applications

3.1 Utilisation des types de données

Le paragraphe 2.3.1 de la CEI 61131-3 offre vingt types élémentaires de données. L'utilisateur peut aussi définir de nouveaux types de données, comme il est décrit en 2.3.3 de la CEI 61131-3, quand c'est nécessaire pour satisfaire aux besoins de l'application. Tous les types de données, y compris les données définies par l'utilisateur, sont disponibles pour être utilisées dans une bibliothèque de types de données, comme décrit en 1.4.3 de la CEI 61131-3. L'utilisateur déclare alors le type de données à utiliser pour chaque variable.

Il convient que le choix d'un type pour une variable soit adapté à la plage de valeurs possibles et aux opérations à réaliser sur la variable. Par exemple:

– si une variable ne peut prendre que les valeurs 0 ou 1, et si elle n'est manipulée que par des opérations booléennes, le type BOOL sera alors choisi;

– si un programme d'automate programmable doit compter quelque chose et que les quantités attendues sont dans une plage de 0 à 1 000, on ne peut pas utiliser une variable de type SINT ou USINT, car leur plage de valeurs va de –128 à +127 pour SINT et de 0 à 255 pour USINT. Un type de données raisonnable dans ce cas sera UINT. Ce type fournit une plage de valeurs suffisante et l'utilisation d'un nombre entier sans signe indique clairement qu'on n'attend pas de valeurs négatives.

3.1.1 Initialisation des variables versus initialisation des types

Toutes les variables d'un programme conforme à la CEI 61131-3 doivent être initialisées, soit explicitement par la programmation, soit implicitement par des mécanismes par défaut définis dans la norme. Il ne doit, en principe, jamais y avoir de variables non initialisées. Pour rendre plus facile la déclaration des variables, tous les types élémentaires ont des valeurs initiales par défaut spécifiées dans la norme. Si l'utilisateur ne spécifie pas d'initialisation d'une variable, cette variable aura la valeur initiale par défaut. La plupart des valeurs initiales par défaut sont la représentation de la valeur zéro pour ce type de données.

La CEI 61131-3 permet aussi à l'utilisateur de spécifier la valeur initiale par défaut pour les types définis par l'utilisateur. Par exemple le type déclaré par:

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

Toute variable déclarée de ce nouveau type «TempLimit» sera initialisée avec une valeur par défaut de 250.0 au lieu de 0.0 comme il aurait été normal pour une donnée REAL (nombre réel). De cette façon, dans les déclarations ci-dessous, la variable «Boiler MaxTemperature» prend la valeur initiale 250,0 alors que la variable «PipeMaxTemperature» prend la valeur

– textual expressions can only be used in the ST language, not the LD or FBD language.

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

2.4.3.2 Inter-system portability

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

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

3 Application guidelines

3.1 Use of data types

Subclause 2.3.1 of IEC 61131-3 offers twenty elementary data types. The user can also define new data types, as described in 2.3.3 of IEC 61131-3, as necessary to meet the data representation needs of the application. All data types, including user defined types, are made available for use in a library of data types as described in 1.4.3 of IEC 61131-3. The user then declares the data type to be used for each variable.

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

– if a variable can only hold the values 0 or 1 and is only to be operated on by Boolean operations, then the elementary type BOOL should be chosen;

– if a programmable controller program has to count something and the counts are expected to be in the range from zero to one thousand, a variable of type SINT or USINT cannot be used, since their value ranges only extend from –128 to +127 for SINT and from 0 to 255 for USINT. A reasonable data type for this purpose would be UINT. This has a sufficient value range and the usage of an unsigned integer type also makes it clear that negative values are not expected.

3.1.1 Type vs. variable initialization

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

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

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

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

initiale 0.0. Si la valeur 0 n'est pas une température maximale raisonnable pour le tuyau, sa valeur correcte doit être établie avant la première utilisation de cette variable. Omettre cela poserait des problèmes. Dans le présent exemple, la température maximale de la chaudière est initialisée avec la valeur adéquate dans sa valeur initiale par défaut. Il est inutile de fixer cette valeur avant la première utilisation, ce qui simplifie grandement les programmes des automates programmables et augmente la fiabilité du logiciel.

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

3.1.2 Utilisation des types à liste de valeurs ou à sous-plage de valeurs

Une donnée de type liste limite les valeurs des variables de ce type à un ensemble de valeurs identifiées par des identificateurs définis par l'utilisateur. Considérons, à titre d'exemple

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

On a ici défini un nouveau type «Color». Il ne peut prendre que trois valeurs: Red, Green ou Blue (rouge, vert ou bleu). La norme ne dit pas quelles valeurs numériques ces valeurs énumérées prendront. Il n'existe pas non plus de fonctions de conversion entre les types à listes de valeurs et le type intégral. Les valeurs doivent simplement être distinctes et reproductibles. L'affectation d'une valeur à la variable «brickColor» n'est possible que si on utilise une des valeurs de couleur définies. Toutes les autres valeurs sont signalées comme étant incorrectes.

La CEI 61131-3 fournit des mécanismes pour définir les types à liste de valeurs ou à sous-plage de valeurs. Ces types n'ajoutent pas de nouvelles fonctionnalités à un programme, mais leur utilisation peut au moins rendre un programme plus lisible et est ainsi une aide à la documentation. Dans certains systèmes il peut aussi y avoir une sorte de vérification des valeurs pour des variables de ce type pendant toute la durée du programme. Ce comportement est en général connu sous le nom de «vérification des plages de valeurs».

3.1.3 Utilisation des données BCD

Il convient que les utilisateurs soient avertis du fait que «BCD» n'est pas un type de données dans la CEI 61131-3. BCD représente plutôt une option de codage pour les chaînes de bits dans les types BYTE (demi-mot), WORD (mot), DWORD (mot double) et LWORD (mot long), où ces types de données sont respectivement représentés par 2, 4, 8, ou 16 chiffres BCD. C'est la reconnaissance du fait que BCD est rarement utilisé dans les systèmes modernes, sauf pour le transfert de données dans une chaîne de bits à partir de dispositifs extérieurs ou vers ces dispositifs tels qu'une unité d'affichage multi-segments ou des commutateurs à mollettes.

Comme les systèmes conformes ne sont pas obligés de supporter l'arithmétique BCD, les données codées en BCD doivent être converties dans un des types entiers (SINT, INT, LINT, USINT, UINT, UDINT ou ULINT) en utilisant une des fonctions de conversion BCD_TO_** définies en 2.5.1.5.1 de la CEI 61131-3, afin de pouvoir être manipulées par des opérations arithmétiques. De façon similaire, les fonctions de conversion **_TO_BCD sont fournies pour convertir des données entières sous la forme de données BCD pour le transfert vers des dispositifs extérieurs.

Il convient que les utilisateurs soient avertis des erreurs potentielles qui peuvent être provoquées pendant le codage des données BCD.

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

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

3.1.2 Use of enumerated and subrange types

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

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

Here a new type “Color” is defined. It may only have three values: Red, Green or Blue. The standard does not state which numerical values these enumerated values should have. There is also no conversion function to and from enumerated types to integral types. The values only have to be distinct and reproducible. An assignment of a value to the variable “brickColor” is possible only if one of the defined color values is used. All other values are flagged as errors.

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

3.1.3 Use of BCD data

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

Since compliant systems are not required to support BCD arithmetic, BCD encoded data must be converted to one of the integer types (SINT, INT, DINT, LINT, USINT, UINT, UDINT or ULINT) using one of the BCD_TO_** conversion functions defined in 2.5.1.5.1 of IEC 61131-3, in order to be manipulated arithmetically. Similarly, the **_TO_BCD functions are provided to convert integer data to BCD encoded from for transfer to external devices.

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

1) Comme il n'existe pas de codage BCD normalisé pour le signe «moins», l'utilisation de nombres négatifs en entrée pour une conversion **_TO_BCD peut provoquer une erreur de conversion.

2) La plage des valeurs d'une variable entière est plus grande que la plage d'une variable chaîne de bits codée en BCD avec le même nombre de bits. Par exemple, la plage d'une variable de type SINT est (–128.. 127), alors que la plage des nombres que l'on peut coder dans une chaîne de bits de type STRING est seulement (0..99).

Les exemples de la figure 4 montrent les précautions qui peuvent être nécessaires pour éviter ces erreurs. Dans cet exemple, le bloc fonctionnel BCD_DIFF fait la différence entre deux entrées codées sur deux bits THUMB1 et THUMB2 et le résultat sorti est une chaîne de bits de deux chiffres BCD plus un signe booléen qui peut servir, par exemple, à amener l'affichage BCD d'un signe plus sur deux chiffres.

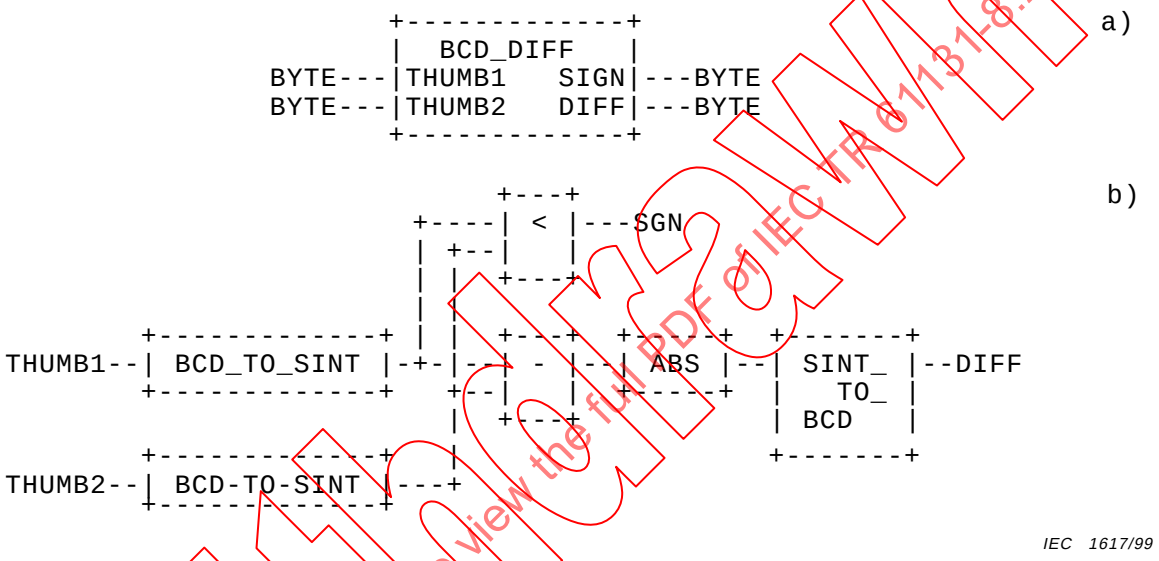


Figure 4 – Bloc fonctionnel BCD_DIFF
a) Interface externe,
b) Corps

Une autre solution à ce problème serait de définir un nouveau type de données structuré contenant le signe et la donnée codée en BCD, tel que SBCD_BYTE illustré à la figure 5a). Des fonctions, telles que SBCD_DIFF illustrées aux figures 5b) et 5c), pourraient alors être définies pour fournir les valeurs de ce nouveau type comme résultat de leur exécution.

1) Since no standard BCD encoding is defined for the “minus” sign, the use of a negative number as an input to a **_TO_BCD conversion may cause a conversion error.

2) The range of an integer variable is larger than the range of a BCD-encoded bit string variable with the same number of bits. For instance, the range of a variable of type SINT is (–128..127), whilst the range of numbers that can be encoded in a bit string of type BYTE is only (0..99).

The example shown in figure 4 illustrates precautions that may be necessary to avoid these errors. In this example, the function block BCD_DIFF takes the difference between two 2-digit BCD-encoded inputs THUMB1 and THUMB2 and outputs the result as a two-digit BCD encoded bit string plus a Boolean sign, which could be used, for example, to drive a “two digit plus sign” BCD display.

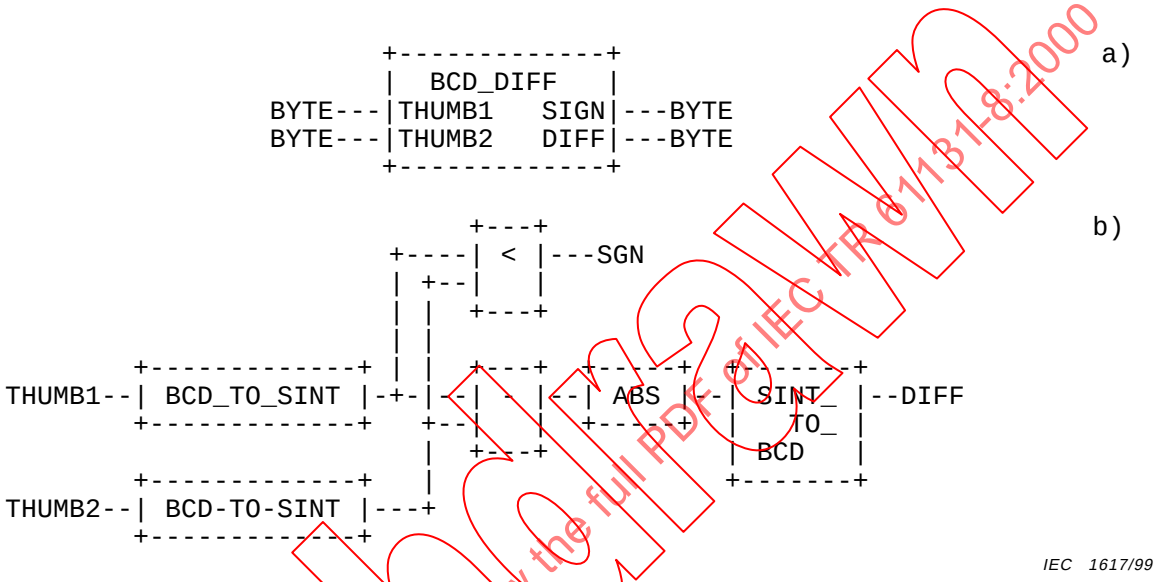


Figure 4 – Function block BCD_DIFF
a) External interface,
b) Body

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

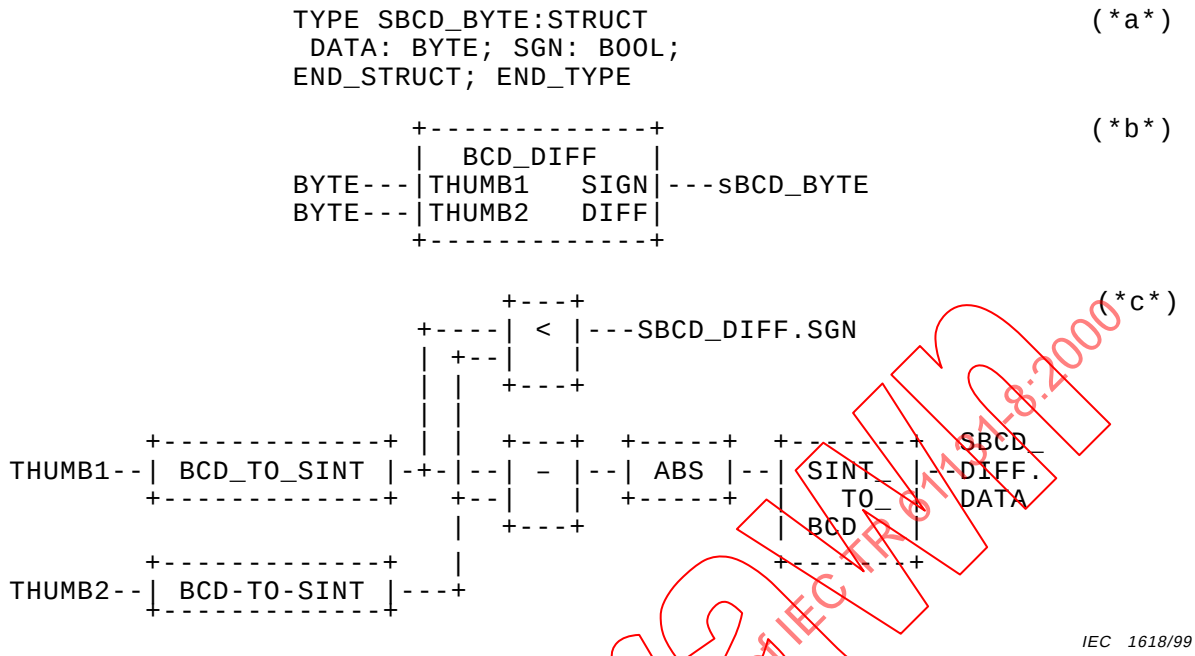


Figure 5 – Bloc fonctionnel SBCD_DIFF
a) Définition du type de données structurées SBCD_BYTE,
b) Interface externe,
c) Corps

3.1.4 Utilisation des types REAL

Le type REAL (nombre réel) décrit en 2.3.1 de la CEI 61131-3 peut servir pour exprimer la majorité des valeurs décimales telles que les valeurs de consigne de commande de boucle et les valeurs de processus. Le type de données REAL offre une très grande plage de valeurs de $\pm 10^{\pm 38}$, avec une précision de 1 pour 2^{23}, soit 1 pour 8388608.

Quand une plage de valeurs plus large ou une meilleure précision sont nécessaires, on doit utiliser le format de 64 bits (long) LREAL avec une plage de $\pm 10^{\pm 308}$ et une précision de 1 pour 2^{52}.

NOTE 1 Dans certains algorithmes, les erreurs d'arrondis peuvent être amplifiées par les calculs effectués. On peut avoir besoin, pour éviter ces erreurs, de types de données dont la précision est plus grande qu'il semblait a priori.

NOTE 2 Voir 4.2.1 pour des considérations supplémentaires lors de la mise en oeuvre de données de type REAL.

3.1.5 Utilisation des données de type chaîne de caractères

Le type de données STRING (chaîne) fournit un stockage pour des données littérales de longueur variable, ce qui est nécessaire dans la majorité des programmes, par exemple pour conserver les identificateurs des processus, les noms des formules, les codes de sécurité des opérateurs.

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

       +-------------+                                         *b*)
       |  BCD_DIFF   |
BYTE---|THUMB1   SIGN|---sBCD_BYTE
BYTE---|THUMB2   DIFF|
       +-------------+

                +---+                                          *c*)
              +-| < |---SBCD_DIFF.SGN
              | +--|   |
              | |  +---+
              | |
       +------------+ | |  +---+  +-----+  +-------+  SBCD_
THUMB1--| BCD_TO_SINT |-+-|--| - |--| ABS |--| SINT_ |--DIFF.
       +------------+   +--|   |  +-----+  |   TO_ |  DATA
                         |  +---+           | BCD   |
       +------------+    |                  +-------+
THUMB2--| BCD-TO-SINT |---+
       +------------+
```

IEC 1618/99

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

3.1.4 Use of REAL data types

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

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

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

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

3.1.5 Use of character string data types

The STRING data type provides storage for variable length textual data, which is required in the majority of application programs, for example for holding processes batch identifiers, recipe names, operator security codes.

La CEI 61131-3 donne aussi des dispositions pour définir des caractères non imprimables dans une chaîne de caractères. Cela est souvent nécessaire quand on construit un message pour un dispositif externe. Par exemple, pour mettre en forme un compte rendu, il peut être nécessaire d'inclure des «alimentations papier» et d'autres caractères de commande similaires dans un message à soumettre à l'impression.

3.1.6 Utilisation des types date et heure

La CEI 61131-3 fournit un certain nombre de types de données pour gérer l'heure du jour, la date et la durée. Enregistrer l'heure à laquelle les activités surviennent, mesurer la durée d'un processus et déclencher des actions à l'instant voulu pendant la journée ou à une date particulière sont des éléments typiques et, dans certains cas, essentiels de la plupart des programmes d'application des processus de production et de fabrication.

L'usage du temps inclut typiquement ce qui suit.

a) Une définition adéquate de la durée d'une phase d'un processus, par exemple pour un traitement thermique, où le temps de recuisson de certains matériaux est critique.

b) L'enregistrement de la date et de l'heure des conditions d'alertes pour procéder à des audits et aussi à des fins de maintenance.

c) Une mise en marche contrôlée d'un processus selon l'heure du jour, par exemple pour initialiser le préchauffage d'une cuve de réacteur avant le premier poste de la semaine.

d) L'enregistrement de la date de calibrage des données analogiques critiques en entrée, de façon que le système puisse prévenir quand un recalibrage est nécessaire.

e) L'enregistrement de l'heure de coupures d'alimentation en énergie et de celle de la reprise, afin de calculer la durée totale des pannes. Cela peut aussi servir à une application pour définir une stratégie contre les défauts de l'alimentation en énergie. Par exemple, si la coupure ne dure que quelques minutes, l'application peut être capable de continuer car les cuves sont toujours chaudes, alors qu'à la suite d'une longue coupure, l'application doit arrêter le processus et faire le nécessaire pour mettre l'usine dans un état de sécurité.

   NOTE Dans cet exemple, on suppose que l'automate programmable est capable de retenir la date et l'heure des coupures d'alimentation en énergie dans une mémoire non effaçable.

f) La définition de délais pour l'exécution de certaines opérations. Par exemple, si la communication d'une transaction avec un dispositif série n'est pas terminée au bout d'un certain temps, on considère que l'opération n'est pas réussie.

Les types de données TIME, TIME_OF_DAY, DATE et DATE_AND_TIME, peuvent être utilisés dans des expressions avec les opérateurs numériques ADD (+), SUB (–), MUL (*), DIV (/) et aussi avec la fonction de concaténation CONCAT. Dans la figure 6, on donne un exemple d'un tel usage.

```
ProcessDuration := phaseDuration * phaseCount;
endTime         := startTime = ProcessD-uration;
endDateAndTime  := CONCAT (todayDate, endTime);
```

IEC 1619/99

Figure 6 – Exemple en littéral structuré (ST) d'utilisation de données de type heure

Il y a également des fonctions de conversion de type pour toutes les manipulations nécessaires entre les dates, les heures du jour et les durées. Par exemple, il est possible de déduire TIME_OF_DAY à partir d'une variable DATE_AND_TIME.

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

3.1.6 Use of time data types

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

Typical usage of time includes the following.

a) Accurate definition of the duration of a process phase, for example, in heat treatment where the annealing time of some materials is critical.

b) Recording the date and time of alarm conditions for process audit and maintenance purposes.

c) Controlled switch-on of a process according to the time of day, e.g. to initiate pre-heating of a reactor vessel before the first shift of the week.

d) Recording the calibration date of critical analogue inputs so that the system can warn when recalibration is required.

e) Recording the times of power failure and power resumption, to calculate the down-time duration. This can be used with an application to define a power-fail strategy. For example, if power fails for a few minutes, the application may be able to continue because the process vessels are still warm, whereas, after a long power-failure, the application should abort the process and take whatever action is necessary to put the plant into a safe state.

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

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

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

```
ProccessDuration := phaseDuration * phaseCount;
endTime          := startTime = ProcessD-uration;
endDateAndTime   := CONCAT (todayDate, endTime);
```

IEC 1619/99

Figure 6 – Structured text (ST) example of time data type usage

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

3.1.7 Utilisation des variables multi-éléments

Il existe des dispositions de la CEI 61131-3 pour des variables multi-éléments («agrégats»), incluant des tableaux et des structures. Les tableaux sont utilisés dans une grande variété de programmes; leur utilisation peut éviter des répétitions de codes et rend dans bien des cas le programme plus facile à comprendre. La figure 7 donne un exemple de l'utilisation d'une variable tableau. Dans cet exemple, «speeds» est un tableau des vitesses de lignes permises, «lineState» est un nombre entier (INT) donnant l'état de la ligne tel que 0 (zéro) pour une ligne coupée, et de 1 à 3 pour des vitesses graduellement croissantes.

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

IEC 1620/99

Figure 7 – Exemple d'utilisation de tableau

3.2 Importation et exportation des données

Il y a diverses méthodes de transfert de données de et vers les fonctions, les blocs fonctionnels et les programmes.

Les méthodes les plus restrictives d'accès aux données sont celles des fonctions. Dans les fonctions, seule est permise la lecture de variables en entrée, la fonction doit renvoyer une valeur unique, qui peut être un agrégat (tableau, chaîne ou structure). Les fonctions n'ont pas accès aux variables définies globalement, ni aux variables en entrée ou sortie (voir 3.2.2), ni à des variables représentées directement. Un des autres objectifs de conception de la norme est qu'il ne faut pas de variables statiques, ce qui signifie qu'une fonction ne peut pas, d'un appel à un autre, conserver de valeurs entrées ou calculées. Chaque invocation recevra un ensemble de variables locales nouvellement initialisées, éventuellement à zéro si la fonction n'en a pas décidé autrement lors de sa définition.

Ces restrictions garantissent que la fonction ne causera jamais d'effets de bord. Elle ne peut modifier aucune autre variable et elle ne dépend que du jeu de paramètres en entrée pour l'invocation en cours.

Dans le corps d'un bloc fonctionnel, il est permis de lire des variables en entrée, de lire ou d'écrire des variables en sortie. Les blocs fonctionnels peuvent aussi avoir des variables en entrée et en sortie, qui peuvent être lues ou écrites. Toutes les variables globales définies dans une configuration, une ressource ou un programme à l'aide du mot clé «VAR_GLOBAL» sont accessibles de l'intérieur du bloc fonctionnel, si on a redéclaré ces variables globales à l'aide du mot clé «VAR_EXTERNAL.» Un bloc fonctionnel n'est cependant pas autorisé à accéder à une variable représentée directement, sauf au moyen d'un nom défini globalement et permettant des alias et que l'on peut référencer dans une déclaration «VAR_EXTERNAL». Les valeurs des variables représentées directement peuvent être soumises à un bloc fonctionnel en tant que paramètres d'entrée à partir d'un programme extérieur à ce bloc fonctionnel. Cela est aussi vrai pour les sorties d'un bloc fonctionnel, qui peuvent être copiées dans des variables représentées directement dans un programme. Un bloc fonctionnel peut déclarer des stockages statiques qu'il peut utiliser pour sauver des données à utiliser lors de l'invocation suivante. L'exécution d'un bloc fonctionnel peut donc avoir de nombreux effets sur les données, y compris les données externes, qui peuvent être essentielles pour l'exploitation du bloc fonctionnel.

Les programmes ont le même accès aux données que les blocs fonctionnels, mais ils peuvent en plus utiliser les variables représentées directement et ils peuvent contenir des déclarations de variables globales et des chemins d'accès (voir 2.5.3 de la CEI 61131-3). Ces variables doivent être déclarées dans le programme et peuvent être ensuite utilisées librement.

3.1.7 Use of multi-element variables

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

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

IEC 1620/99

Figure 7 – Example of array usage

3.2 Data import and export

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

The most restricted methods to access data are defined for functions. Inside functions, only reading of input variables is allowed and a function has to return a single value, which may be of an aggregate type (e.g. array, string or structure). Functions do not have any access to globally defined variables, nor can they use input/output variables (see 3.2.2), nor may they access directly represented variables. As another design goal the standard requires functions to have no static variables. This means a function cannot save any of its computed or input values from one invocation to the next. Each invocation will receive a set of fresh initialized local variables, possibly with values of zero if not stated differently by the function definition.

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

Within a function block body, the reading of input variables and reading or writing of output variables are permitted. Function blocks also may have input/output variables, which can be read or written. All global variables that were defined within a configuration, resource or program by use of the "VAR_GLOBAL" keyword can be accessed from the inside of the function block, if these global variables are redeclared in the function block definition by use of the "VAR_EXTERNAL" keyword. However, a function block is not allowed to access any directly represented variables, except by means of a globally defined aliasing name that can be referenced in a "VAR_EXTERNAL" declaration. Nevertheless, the values of directly represented variables can be passed to a function block as input parameters from the program outside that function block. The same holds for output values from a function block, which can be copied to directly represented variables in a program. A function block may declare static storage that it can use to save any data from one invocation to the next. Function block execution can therefore have many effects upon data, including external data, which may be essential to the operation of the function block.

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

3.2.1 Variables globales et variables externes

Les variables globales peuvent être définies et initialisées dans une configuration, dans une ressource ou dans un programme à l'aide du mot clé «VAR_GLOBAL». Chaque programme ou chaque bloc fonctionnel qui doit accéder à une ou plusieurs de ces variables doit redéclarer ces variables en utilisant le mot clé «VAR_EXTERNAL».

L'exemple suivant montre comment on définit la variable «TEMP» en dehors d'un bloc fonctionnel et comment on y accède dans un bloc fonctionnel.

```
(* Dans une CONFIGURATION, RESSOURCE ou PROGRAMME *)
        VAR_GLOBAL TEMP: INT; END_VAR

(* Accès à partir de l'intérieur d'un BLOC FONCTIONNEL *)
        VAR_EXTERNAL TEMP: INT; END_VAR
        ...
        TEMP:=...;
```

Comme les déclarations de variables globales incluent les variables représentées directement, on peut déclarer des alias. Ces alias peuvent être référencés par des déclarations de variables externes dans des blocs fonctionnels et dans des programmes.

Dans l'exemple suivant, on déclare qu'une variable globale «TEMP» est de type INT placée dans un mot en entrée %IW22. Dans un bloc fonctionnel, le nom de variable TEMP est utilisé comme alias pour ce mot en entrée. Un bloc fonctionnel peut ne pas lire directement le mot en entrée %IW22, mais il peut à la place utiliser l'alias.

```
(* Dans une CONFIGURATION, RESSOURCE ou PROGRAMME *)
        VAR_GLOBAL TEMP AT %IW22: INT; END_VAR

(* Accès à partir de l'intérieur d'un BLOC FONCTIONNEL *)
        VAR_EXTERNAL TEMP: INT; END_VAR
        ...
        ...:= TEMP;
```

3.2.2 Variables en entrée/sortie (VAR_IN_OUT)

Les variables en entrée/sortie sont une espèce spéciale de variable que l'on n'utilise qu'avec les blocs fonctionnels et les programmes. Elles ne représentent aucune donnée directement, mais elles référencent d'autres données du type approprié. On les déclare à l'aide du mot clé «VAR_IN_OUT». Les variables en entrée/sortie peuvent être lues ou écrites.

Dans un bloc fonctionnel, les variables en entrée/sortie permettent d'accéder à la valeur originale d'une variable au lieu d'une copie de la valeur contenue par cette variable. Considérons, par exemple, le bloc fonctionnel ACCUM décrit à la figure 8a). A chaque invocation d'une instance du bloc fonctionnel, la valeur en cours de la variable «X» est ajoutée à celle de la variable en entrée/sortie «A». Si une instance de ce bloc fonctionnel est déclarée et invoquée comme le montre la figure 8b), la variable «ACC» sera elle-même augmentée du produit «X1*X2» à chaque invocation d'une instance du bloc fonctionnel «SUM_PROD». Il est inutile de copier la valeur de sortie de AAC1.A dans la variable «ACC». De façon similaire, dans la figure 8c), la variable «ACC» sera augmentée de la somme des produits «X1*X2 + X3*X4» à chaque invocation d'une instance du bloc fonctionnel «SUM_2_PROD».

Comme les sorties d'une fonction n'ont pas de stockage permanent qui leur est associé, la sortie d'une fonction ne peut pas être utilisée comme paramètre dans une variable VAR_IN_OUT, comme le montre la figure 8d). Comme on peut écrire dans une variable en entrée/sortie, il s'ensuit que des littéraux, par exemple «2.0», ou des constantes affectées par des déclarations VAR CONSTANT, ne peuvent être assignées en tant que variables en entrée/sortie, comme le montre la figure 8e).

3.2.1 Global and external variables

Global variables can be defined and initialized within a configuration, resource or program by use of the “VAR_GLOBAL” keyword. Each program or function block that needs access to one or more of these global variables has to redeclare the variables by use of the “VAR_EXTERNAL” keyword.

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

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

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

In the following example, a global variable “TEMP” is declared to be of type INT located at input word %IW22. Within a function block, the variable name “TEMP” is used as an alias for this input word. A function block may not directly read the input word %IW22 but may indirectly use the alias instead.

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

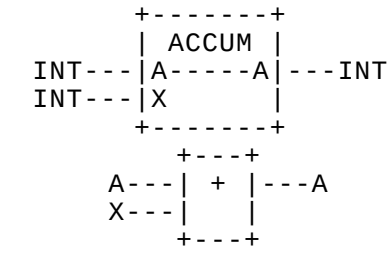
3.2.2 Input/output (VAR_IN_OUT) variables

Input/output variables are a special kind of variable used only with function blocks and programs. They do not represent any data directly but reference other data of the appropriate type. They are declared by use of the “VAR_IN_OUT” keyword. Input/output variables may be read or written to.

Inside a function block, input/output variables allow access to the original instance of a variable instead of a copy of the value contained in the variable. Consider, for instance, the function block ACCUM illustrated in figure 8a). Upon each invocation of an instance of the function block, the current value of the input “X” is added to the input/output variable “A”. If an instance of this function block is declared and invoked as shown in figure 8b), the variable “ACC” itself will be augmented by the product “X1*X2” at each invocation of an instance of the function block “SUM_PROD”. There is no need to copy the output value from ACC1.A back into the variable “ACC”. Similarly, in figure 8c), the variable “ACC” will be augmented by the sum of products “X1*X2 + X3*X4” at each invocation of an instance of the function block “SUM_2_PROD”.

Since the output of a function does not have permanent storage associated with it, the output of a function cannot be used as a VAR_IN_OUT parameter, as illustrated in figure 8d). Since an input/output variable may be written to, it follows that integrals, e.g. “2.0” or constants that have been allocated in VAR CONSTANT declarations, cannot be assigned as input/output variables, as illustrated in figure 8e).

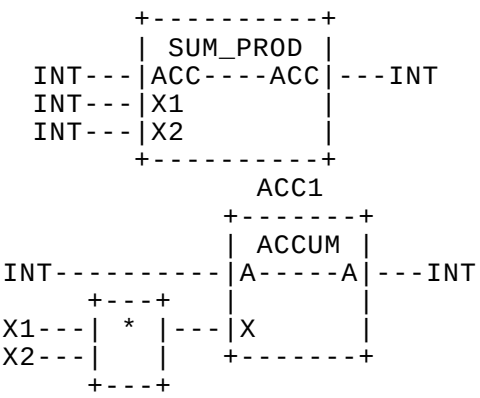
a)




```
FUNCTION_BLOCK ACCUM

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

b)

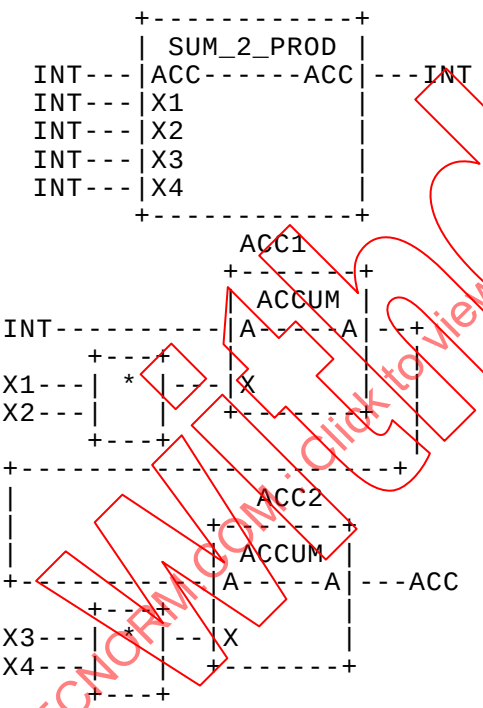



```
FUNCTION_BLOCK SUM_PROD

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


END_FUNCTION_BLOCK
```

c)

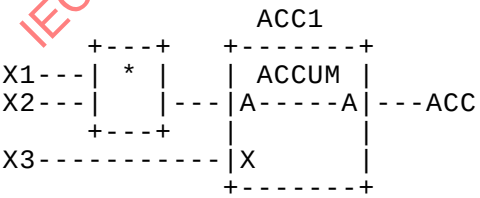



```
FUNCTION_BLOCK SUM_2_PROD

VAR_INPUT X1,X2,X3,X4:INT;
END_VAR
VAR_IN_OUT ACC:INT; END_VAR
VAR ACC1,ACC2:ACCUM;
END_VAR


END_FUNCTION_BLOCK
```

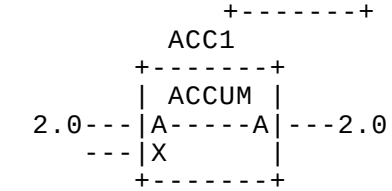
d)



USAGE INTERDIT:

L'entrée/sortie A n'est pas une variable ni un nom de bloc fonctionnel

e)



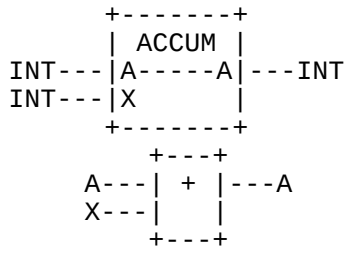
USAGE INTERDIT:

L'entrée/sortie A n'est pas une variable ni un nom de bloc fonctionnel

IEC 1621/99

Figure 8 – Exemples d'utilisation de VAR_IN_OUT

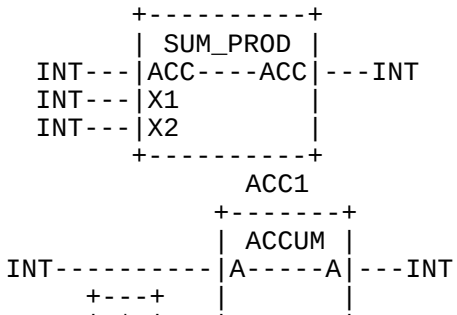
a)




```
FUNCTION_BLOCK ACCUM

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

b)

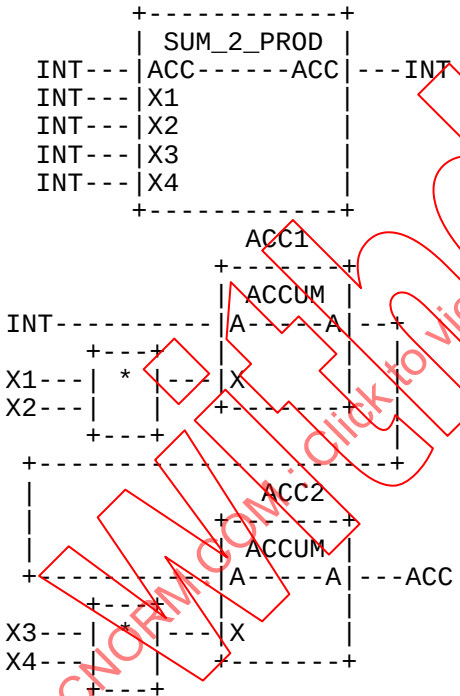



```
FUNCTION_BLOCK SUM_PROD

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


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

c)

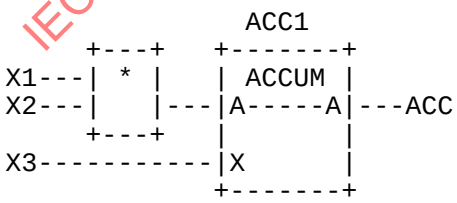



```
FUNCTION_BLOCK SUM_2_PROD

VAR_INPUT X1,X2,X3,X4:INT; END_VAR
VAR_IN_OUT ACC:INT ; END_VAR
VAR ACC1,ACC2:ACCUM; END_VAR


ACC1(A:= ACC, X:= X1 * X2);
ACC2(A:= ACC, X:= x3 * X4);
END_FUNCTION_BLOCK
```

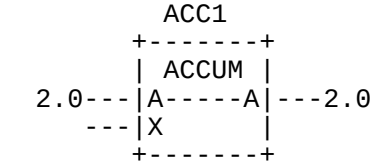
d)



ILLEGAL USAGE:

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

e)



ILLEGAL USAGE:

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

IEC 1621/99

Figure 8 – Examples of VAR_IN_OUT usage

3.3 Utilisation des blocs fonctionnels

3.3.1 Types et instances de blocs fonctionnels

Un *type de bloc fonctionnel* est une unité d'organisation de programmes (POU = program organisation unit). Cette POU décrit les variables en entrée et en sortie ainsi que les données locales de la zone, pour des *instances* du bloc fonctionnel. Il contient en outre les règles de traitement de ces données lorsqu'une instance du bloc fonctionnel est invoquée. Les variables ne peuvent pas être lues à partir du bloc fonctionnel ou écrites dans le bloc fonctionnel lui-même, il n'est pas non plus possible d'invoquer le type de bloc fonctionnel lui-même; ces opérations sont réservées aux instances du bloc fonctionnel.

On peut utiliser les *instances multiples* de blocs fonctionnels basées sur cette déclaration de type. Les instances individuelles sont indépendantes les unes des autres. Chaque instance de bloc fonctionnel a un identificateur unique (le nom de l'instance) et une zone de données privées utilisée en entrée, en sortie et en variables internes pour cette instance de bloc fonctionnel. On peut accéder à une instance de bloc fonctionnel et l'invoquer par son nom d'instance.

Ces qualités des blocs fonctionnels sont relatives à une programmation orientée objet (OOP = object oriented programming). Le type de bloc fonctionnel est similaire à une *classe*, qui définit la structure de données ainsi que les méthodes de traitement dans le corps du bloc fonctionnel. Les objets individuels sont représentés par les zones de données privées de chaque instance individuelle du bloc fonctionnel. Ces données ne peuvent être modifiées que d'une manière contrôlée, à partir de l'extérieur du corps du bloc fonctionnel. Cela confirme les principes d'ingénierie logicielle d'encapsulage et de masquage de l'information, un élément clé de l'OOP.

Les instances typiques de blocs fonctionnels sont les minuteurs ou les compteurs, qui gardent leurs valeurs d'une invocation à une autre et qui déterminent si on a ou non atteint une valeur finale.

L'accès contrôlé à des dispositifs partagés constitue un autre domaine d'intérêt pour l'utilisation des blocs fonctionnels. Dans ce cas, une instance unique du bloc fonctionnel a le contrôle exclusif de ce dispositif et joue le rôle de sémaphore, le dispositif ne pouvant être atteint que si l'on invoque l'instance correspondante du bloc fonctionnel.

L'avantage qu'il y a à utiliser les instances de blocs fonctionnels est que la fonctionnalité associée à une structure de données définie ne doit être déclarée qu'une seule fois et peut être utilisée indépendamment dans plusieurs instances dans le programme d'un automate programmable. Ce «prototype» est conservé dans le type de bloc fonctionnel, et il peut resservir autant de fois qu'il est nécessaire quand on déclare les instances de ce type. L'utilisateur a donc la garantie qu'il n'y a pas d'erreurs dans aucune des instances de blocs fonctionnels tant qu'il n'y a pas d'erreurs dans le type de bloc fonctionnel associé.

Etant donné que le jeu complet de données d'état de l'instance en cours est facilement accessible pour une surveillance et une correction en ligne, les instances de blocs fonctionnels peuvent être une aide aux essais et au débogage.

Les instances de blocs fonctionnels sont un accessoire spécial des blocs fonctionnels tels qu'ils sont définis dans la CEI 61131-3. Il n'y a pas d'équivalent dans les langages de programmation procéduraux comme Pascal ou C.

L'instanciation des blocs fonctionnels est une extension des possibilités des automates programmables actuels. Dans de nombreuses installations actuelles, il n'y a qu'un nombre fixe d'instances de chaque type disponible de blocs fonctionnels, et il n'est pas possible de créer des instances supplémentaires. Les blocs fonctionnels définis par l'utilisateur pour lesquels on peut créer sans limites des instances sont aussi une extension pour la plupart des systèmes en place.

3.3 Use of function blocks

3.3.1 Function block types and instances

A *function block type* is a program organization unit (POU). This POU described 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 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 for 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 Portée des données dans un bloc fonctionnel

Le principe d'encapsulage des données et du masquage des noms de variables discuté dans le précédent paragraphe peut ne pas être familier aux utilisateurs des systèmes d'automates programmables traditionnels. Ce principe s'applique aussi aux instances de blocs fonctionnels qui sont déclarées de la même manière que des variables dans une structure VAR...VAR_END. Les instances de blocs fonctionnels qui apparaissent dans un autre bloc fonctionnel ne sont par conséquent pas visibles en dehors du bloc fonctionnel conteneur, ce qui est contraire aux pratiques de certains systèmes traditionnels d'automates programmables. Ce principe est exprimé en 2.5.2 de la CEI 61131-3 dans la phrase:

> «La portée d'une instance de bloc fonctionnel doit être locale à l'unité d'organisation de programmes dans laquelle figure son instance, sauf s'il est déclaré comme étant global par un bloc VAR_GLOBAL tel que défini en 2.7.1.»

Cette exigence pourrait être considérée comme étant en contradiction avec l'affirmation en 2.5.2 de la CEI 61131-3: «Aucun bloc fonctionnel qui a déjà été déclaré ne peut être utilisé dans une déclaration d'un autre bloc fonctionnel ou d'un programme, comme le montre la figure 3.» Il est cependant clair, si l'on se réfère à la figure 3 de la CEI 61131-3, que cette affirmation se réfère aux *types* de blocs fonctionnels et non aux *instances* de blocs fonctionnels; il n'y a donc pas deux exigences contradictoires.

La figure 9 illustre une application de ce principe. Dans ce cas, une instance nommée FB1 d'un type de bloc fonctionnel FBy surviendra à chaque instance d'un bloc fonctionnel de type FBx. Quand on a deux instances du bloc fonctionnel FBx dans une instance de programme de type A, on crée deux instances séparées et distinctes du bloc fonctionnel de type FBy. Comme on le voit dans la figure 9c), chaque instance constitue une partie de la zone de données privées d'une instance associée de FBx (FB1 et FB2, respectivement) et est en conséquence invisible hors de cette instance.

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 VAR..END_VAR construct. Hence, function block instances that appear inside another function block are not visible outside the containing function block, contrary to the practice in some traditional programmable controller systems. This principle is expressed in 2.5.2 of IEC 61131-3 in the statement:

> "The scope of an instance of a function block shall be local to the program organization unit in which it is instantiated, unless it is declared to be global in a VAR_GLOBAL block as defined in 2.7.1."

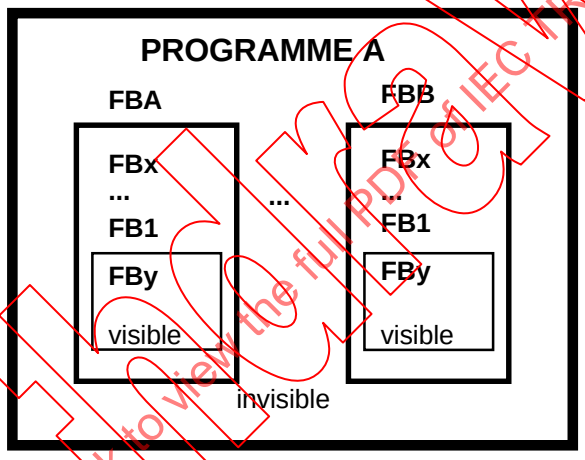
This requirement might be considered to contradict the assertion in 2.5.2 of IEC 61131-3 that "any function block which has already been declared can be used in the declaration of another function block or program as shown in figure 3." However, it is clear by reference to figure 3 of IEC 61131-3 that this latter assertion refers to function block *types*, not function block *instances*; hence, the two requirements are not contradictory.

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

a)
```
FUNCTION_BLOCK FBx
 VAR FB1: FBy; END_VAR   (* FBy est un type de bloc fonctionnel *)
...
 FB1(....);               (* Invoquer l'instance FB1 *)
...
END_FUNCTION_BLOCK
```

b)
```
PROGRAM A
 VAR FBA: FBx;  (* Deux instances de type FBx *)
     FBB: FBx;  (* Contenant chacune une instance FB1 de type FBy *)
 END_VAR;
...
 FBA(....)           (* Invoquer l'instance FBA *)
 FBB(....)           (* Invoquer l'instance FBB *)
...
END_PROGRAM
```

c)



IEC 1622/99

NOTE La suppression des détails inutiles est signalée par le signe (...)

Figure 9 – Masquage des instances du bloc fonctionnel
a) Déclaration du type de bloc fonctionnel contenu
b) Déclaration du type de programme conteneur
c) Visibilité des instances de blocs fonctionnels

3.3.3 Accès aux blocs fonctionnels et invocation des blocs fonctionnels

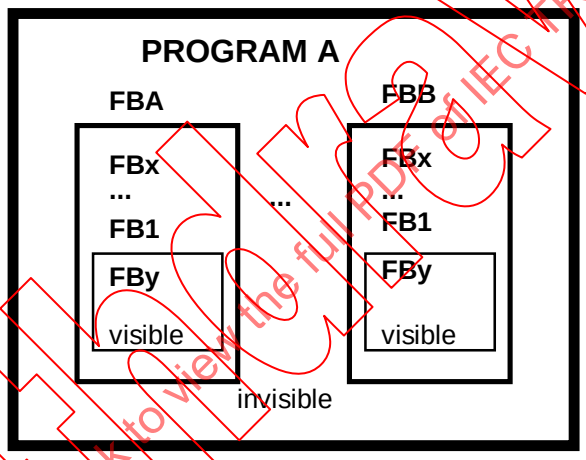
Il y a une différence entre l'accès en lecture à une variable d'une instance d'un bloc fonctionnel et l'*invocation* du bloc fonctionnel lui-même. Lire une variable en sortie d'une instance d'un bloc fonctionnel équivaut à lire la valeur d'un élément d'une variable structurée. Cependant, et contrairement au cas des variables structurées, on n'autorise pas une assignation directe aux variables en sortie du bloc fonctionnel. L'assignation de valeurs à des variables en sortie d'une instance de bloc fonctionnel n'est permise que dans le corps d'une instance de bloc fonctionnel (voir le tableau 32 de la CEI 61131-3).

NOTE Il est en général possible d'invoquer plusieurs fois («affectation multiple») une instance unique d'un bloc fonctionnel dans une POU. Cependant, en fonction de la mise en oeuvre de l'automate programmable, cette possibilité peut être réduite à une seule invocation de chaque instance de bloc fonctionnel de la POU. Une POU qui utilise des invocations multiples d'une même instance de bloc fonctionnel pourrait ne pas être portable vers de telles installations (pour plus de détails, voir 3.7).

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

b)
```
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
```

c)



IEC 1622/99

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

Figure 9 – Hiding of function block instances
a) Declaration of contained function block type
b) Declaration of containing program type
c) Visibility of function block instances

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 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 for more details.

Toutes les valeurs de la zone de données privées d'une instance de bloc fonctionnel persistent d'une invocation à la suivante. La zone de données privées de cette instance peut donc être considérée comme une mémoire de stockage de l'état en cours. A cause de cela, des invocations différentes de la même instance d'un bloc fonctionnel, avec les mêmes variables en entrée, peuvent générer des valeurs différentes dans la zone de données de cette instance.

L'affectation de valeurs aux variables en entrée n'est permise qu'en invoquant une instance du bloc fonctionnel. Cependant, toutes les variables en entrée d'une instance de bloc fonctionnel ne doivent pas recevoir explicitement une valeur pour permettre l'invocation. Cela est vrai car les valeurs de variables non définies ne sont pas possibles pour les raisons suivantes:

– des valeurs par défaut normalisées sont définies pour tous les types de données possibles au moment de l'initialisation des variables;
– la valeur initiale de chaque variable peut être spécifiée lors de sa déclaration;
– toutes les variables gardent leur valeur d'une invocation à la suivante.

NOTE Il n'est actuellement pas possible d'avoir une initialisation spécifique des variables pour des instances séparées d'un bloc fonctionnel (voir B.1.4.3 de la CEI 61131-3).

3.4 Différence entre instance de bloc fonctionnel et fonction

Bien que bloc fonctionnel et fonction soient tous deux des POU (unité d'organisation de programmes), il existe des différences significatives entre bloc fonctionnel et fonction (voir 2.5.1 de la CEI 61131-3):

1) l'invocation d'une fonction a exactement un seul résultat (sauf quand on utilise ENO dans les langages graphiques, voir 2.5.1.2 de la CEI 61131-3);
2) le résultat de l'invocation d'une fonction peut être utilisé comme un opérande dans une expression;
3) une fonction ne peut pas avoir de mémoire privée (c'est-à-dire des informations internes d'état) qui soit modifiée historiquement par les invocations précédentes. Chaque appel avec les mêmes arguments génère donc le même résultat;
4) la portée du nom de la fonction, comme celle du type de bloc fonctionnel, est globale, contrairement à celle du nom d'une instance de bloc fonctionnel.

3.5 Utilisation d'instances de blocs fonctionnels indirectement référencées

Une instance de bloc fonctionnel est référencée à des fins de lecture via ses variables en sortie ou pour invoquer l'instance référencée. Ces opérations peuvent aussi être effectuées sur une instance de bloc fonctionnel dont le nom est passé comme argument d'une POU (unité d'organisation de programmes). Dans ce cas, l'instance de bloc fonctionnel n'est pas directement référencée en utilisant un nom fixe pour l'instance de bloc fonctionnel; au contraire la référence est *indirecte* par le biais d'une variable externe ou en entrée de la POU. Le nom de l'instance du bloc fonctionnel référencé est affecté à une variable appropriée de l'extérieur de la POU dans laquelle il est référencé.

Ce mécanisme permet un accès à différentes instances d'un type de bloc fonctionnel spécifié dans le corps d'un autre bloc fonctionnel ou une invocation de ces différentes instances.

La technique d'utilisation des noms d'instances de blocs fonctionnels comme arguments offre à l'utilisateur de nombreuses possibilités nouvelles en programmation. Ce dispositif rend possible l'accès ou l'invocation de l'instance d'un type de bloc fonctionnel sans avoir besoin de définir l'instance particulière du bloc fonctionnel référencé dans le paragraphe déclaration de la POU qui référence. L'instance référencée peut, en outre, changer d'une invocation, par la POU qui référence, à une autre.

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 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 Currently, it is not possible to have specific initializations of variables for single instances of a function block (see B.1.4.3 of IEC 61131-3).

3.4 Differences between function block instances and functions

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

1) the invocation of a function has exactly one result (except for the use of ENO in graphical languages; see 2.5.1.2 of IEC 61131-3);
2) the result of invoking a function can be used as an operand within an expression;
3) 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;
4) 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 program organization unit (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 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.

On peut trouver des applications utiles de ce mécanisme en liaison avec des problèmes de gestion de plusieurs machines qui ont le même comportement et qui sont chacune représentées par une instance unique de bloc fonctionnel.

3.5.1 Mise en place d'une référence indirecte d'instance de bloc fonctionnel

Les paragraphes 2.5.2, B.1.4.3 et B.1.5.2 de la CEI 61131-3 définissent les mécanismes pour établir une référence indirecte à une instance d'un bloc fonctionnel. Comme le montre la figure 10, la référence du bloc fonctionnel peut être établie comme étant:

a) une variable en entrée;

b) une variable définie dans une déclaration VAR_IN-OUT;

c) une variable externe.

Dans chaque exemple de la figure 10, on montre la définition de l'interface du bloc qui référence une instance de bloc fonctionnel, suivie par un exemple du passage de l'instance du bloc fonctionnel référencé comme faisant partie de l'invocation du bloc fonctionnel qui référence.

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

3.5.1 Establishing an indirect function block instance reference

Subclauses 2.5.2, 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, the function block reference may be established as:

a) an input variable;

b) a variable defined 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.

a)
```
        +--------------+     (*Type de bloc qui référence *)
        |   INSIDE_A   |
TON---|I_TMR  EXPIRED|---BOOL
        +--------------+
 FUNCTION_BLOCK              (* Bloc passant les références *)
        +-----------+        (* Interface externe *)
        | EXAMPLE_A |
 BOOL---|GO     DONE|
        +-----------+
                             (* Corps du bloc fonctionnel *)
       (* Bloc référencé *)
            E_TMR            (* Bloc qui référence *)
          +-----+                  I_BLK
          | TON |              +--------------+
      GO---|IN  Q|              |   INSIDE_A   |
   t#100ms--|PE ET|     E_TMR---|I_TMR  EXPIRED|
          +-----+              +--------------+
END_FUNCTION_BLOCK
```

b)
```
        +--------------+     (* Référencement du type du bloc *)
        |   INSIDE_B   |
TON---|I_TMR----I_TMR|---TON
BOOL--|TMR_GO EXPIRED|---BOOL
        +--------------+
 FUNCTION_BLOCK              (* Bloc de passage de références *)
        +--------------+  (* Interface externe *)
        |   INSIDE_B   |
  BOOL--|TMR_GO EXPIRED|
        +--------------+
                             (* Corps du bloc fonctionnel *)
      (* Bloc référencé *)
            E_TMR            (* Bloc qui référence *)
          +-----+                  I_BLK
          | TON |              +-------------+
          |IN  Q|              |   INSIDE_B   |
   t#100ms---|PT RT|    E_TMR---|I_TMR----I_TMR|
          +-----+      GO------|TMR_GO EXPIRED|---DONE
                               +--------------+
END_FUNCTION_BLOCK
```

c)
```
 FUNCTION_BLOCK               (* Type du bloc qui référence *)
        +--------------+    (* Interface externe *)
        |   EXAMPLE_C  |
  BOOL---|TMR_GO EXPIRED|
        +--------------+
VAR_EXTERNAL X_TMR: TON; END_VAR
END_FUNCTION_BLOCK
 PROGRAM                      (* Programme passant les références *)
                             (* Interface externe *)
        +--------------+
        |   EXAMPLE_C  |
  BOOL---|GO        DONE|---BOOL
        +--------------+
  VAR_GLOBAL X_TMR: TON; END_VAR (* Bloc référencé *)
                              (* Corps du programme *)
            I_BLK            (* Bloc qui référence *)
        +---------------+
        |    INSIDE_C   |
  GO-----|TMR-GO  EXPIRED|---DONE
        +---------------+
END_PROGRAM
```

IEC 1623/99

Figure 10 – Mise en place d'une instance de bloc fonctionnel référencée indirectement
a) Mise en place en tant que variable en entrée
b) Mise en place en tant que variable d'entrée/sortie
c) Mise en place en tant que variable externe

a)
```
        +--------------+     (* Referencing block type *)
        |   INSIDE_A   |
TON---|I_TMR  EXPIRED|---BOOL
        +--------------+
FUNCTION_BLOCK              (* Reference-passing block *)
        +-----------+       (* External interface *)
        | EXAMPLE_A |
 BOOL---|GO     DONE|
        +-----------+
                            (* Function block body *)
      (* Referenced block *)
           E_TMR            (* Referencing block*)
          +-----+                 I_BLK
          | TON |              +--------------+
      GO---|IN  Q|              |   INSIDE_A   |
  t#100ms--|PE ET|     E_TMR---|I_TMR  EXPIRED|
          +-----+              +--------------+
END_FUNCTION_BLOCK
```

b)
```
        +--------------+     (* Referencing block type *)
        |   INSIDE_B   |
TON---|I_TMR----I_TMR|---TON
BOOL--|TMR_GO EXPIRED|---BOOL
        +--------------+
FUNCTION_BLOCK              (* Reference-passing block *)
        +--------------+    (* External interface *)
        |  EXAMPLE_B   |
  BOOL--| GO       DONE|--BOOL
        +---------------+
                            (* Function block body *)
      (* Referenced block *)
            E_TMR           (* Referencing block *)
           +-----+               I_BLK
           | TON |            +--------------+
           |IN  Q|            |   INSIDE_B   |
  t#100ms---|PT RT|    E_TMR---|I_TMR----I_TMR|
           +-----+     GO------|TMR_GO EXPIRED|---DONE
                             +--------------+
END_FUNCTION_BLOCK
```

c)
```
FUNCTION_BLOCK              (* Referencing block type *)
        +--------------+    (* External interface *)
        |  EXAMPLE_C   |
  BOOL---|TMR_GO EXPIRED|
        +--------------+
VAR_EXTERNAL X_TMR: TON; END_VAR
END_FUNCTION_BLOCK
PROGRAM                     (* Reference-passing program *)
                            (* External interface  *)
        +--------------+
        |   EXAMPLE_C  |
  BOOL---|GO        DONE|---BOOL
        +--------------+
  VAR_GLOBAL X_TMR: TON; END_VAR (* Referenced block *)

                            (* Program body *)
           I_BLK            (* Referencing block *)
        +---------------+
        |    INSIDE_C   |
  GO-----|TMR-GO  EXPIRED|---DONE
        +---------------+
END_PROGRAM
```

IEC 1623/99

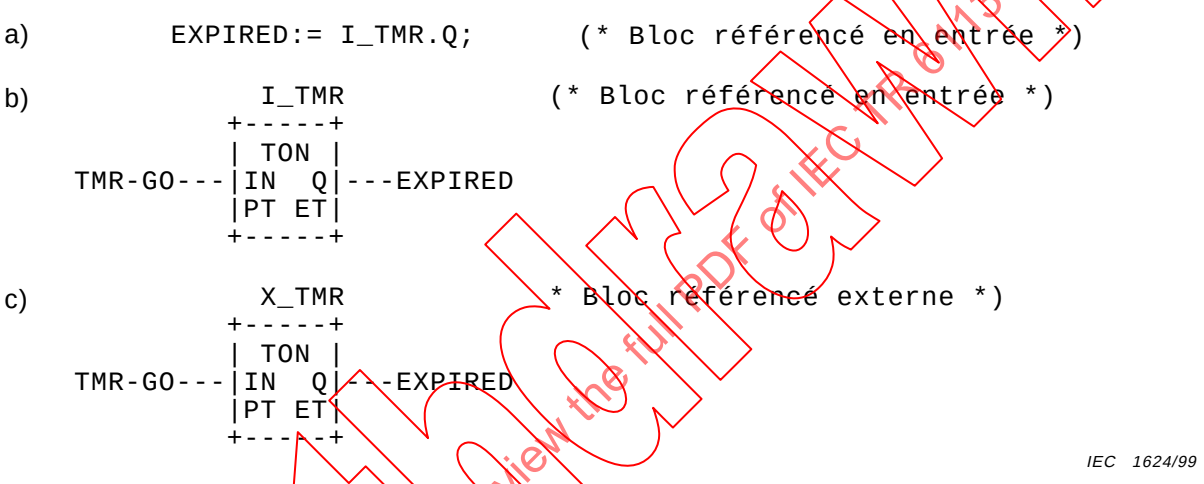
Figure 10 – Establishing an indirectly referenced function block instance
a) Establishment as input variable
b) Establishment as input/output variable
c) Establishment as external variable

3.5.2 Accès aux instances de blocs fonctionnels référencées indirectement

Accéder à une instance de bloc fonctionnel référencée indirectement signifie *lire* (c'est-à-dire utiliser sans modifier) ses variables en sortie. La zone de données privées de l'instance du bloc fonctionnel reste inchangée (voir 2.5.2.1, tableau 32 et 2.5.2.2, point 5 de la CEI 61131-3).

Des exemples de cette utilisation sont donnés aux figures 11a) à 11c), correspondant respectivement aux interfaces et aux déclarations de variables données aux figures 10a) à 10c). Dans chaque cas, le «Q» sortie du bloc fonctionnel référencé TON est passé dans la sortie «EXPIRED» du bloc fonctionnel qui référence.

NOTE On ne peut pas utiliser de représentation graphique pour une instance de bloc fonctionnel référencée indirectement et passée comme paramètre d'entrée. Conformément à 4.1.3 de la CEI 61131-3, cela impliquerait une invocation du bloc fonctionnel qui pourrait provoquer un changement des valeurs des variables de l'instance du bloc fonctionnel référencé. Cela est interdit à une variable en entrée selon 2.5.2.1 de la CEI 61131-3. Cet accès utilisera donc la représentation de la sortie du bloc fonctionnel comme un élément d'une variable structurée, comme le montre la figure 11a).



NOTE Voir la figure 10 pour les définitions de variables et d'interfaces correspondantes.

Figure 11 – Accès à une instance d'un bloc fonctionnel référencée indirectement

3.5.3 Invocation des instances de blocs fonctionnels référencées indirectement

Comme décrit en 2.5.2.2 point 6) de la CEI 61131-3, l'*invocation* d'une instance de bloc fonctionnel référencée indirectement signifie que l'invocation de cette instance de bloc fonctionnel est faite de l'intérieur d'un autre bloc fonctionnel. En utilisant cette propriété, on peut modifier la zone de données privées (et, par là, également les variables en sortie) de cette instance de bloc fonctionnel.

Les règles générales pour une invocation d'une instance de bloc fonctionnel fournies en 2.5.2.1 tableau 32 de la CEI 61131-3, sont valables dans le cas présent:

– assigner une valeur à une variable en entrée de l'instance de bloc fonctionnel référencée indirectement n'est permis que comme faisant partie de l'invocation de ce bloc fonctionnel;

– les variables en sortie de cette instance de bloc fonctionnel ne peuvent pas être modifiées de l'extérieur.

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 2.5.2.1, table 32 and 2.5.2.2, item 5) 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.

NOTE A graphical representation of an indirectly referenced function block instance passed as an *input* parameter cannot be used. According to 4.1.3 of IEC 61131-3, this would imply *invocation* of the function block, which might cause a change in the values of the instance variables of the referenced function block. This is illegal for an input variable as per 2.5.2.1 of IEC 61131-3. Therefore, such access must use the representation of the function block output as an element of a structured variable, as illustrated in figure 11a).

a)
```
EXPIRED := I_TMR.Q;      (* Referenced input block *)
```

b)
```
                I_TMR              (* Referenced input/output block *)
               +-----+
               | TON |
TMR-GO---|IN  Q|---EXPIRED
               |PT ET|
               +-----+
```

c)
```
                X_TMR              (* Referenced external block *)
               +-----+
               | TON |
TMR-GO---|IN  Q|---EXPIRED
               |PT ET|
               +-----+
```

IEC 1624/99

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 2.5.2.2, item 6) of IEC 61131-3, the *invocation* of an indirectly referenced function block instance means the invocation of the function block instance from inside another function block. Using this capability, the private data area (and, therefore, also the output variables) of this function block instance can be modified.

The general rules for an invocation of a function block instance given in 2.5.2.1, table 32 of IEC 61131-3 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.

Comme il est défini en B.1.4.3 et B.1.5.2 de la CEI 61331-3, on peut constituer la référence à l'instance de bloc fonctionnel invoquée par le biais:

– soit d'une variable déclarée dans une déclaration VAR_IN_OUT, comme le montre la figure 11;
– soit d'une variable externe.

Les figures 11b) et 11c) donnent respectivement des exemples de cette utilisation.

Il n'est pas permis d'invoquer une instance indirectement référencée de bloc fonctionnel par le biais d'une variable en entrée, parce qu'une invocation de cette instance de bloc fonctionnel peut changer les valeurs dans la zone de données privées. Les valeurs modifiées peuvent avoir des effets en dehors du bloc fonctionnel invoqué. Ce comportement est interdit pour les variables en entrée. Comme il est noté dans le paragraphe précédent, cela interdit une représentation graphique (qui implique une invocation) des instances de bloc fonctionnel référencées indirectement et constituées par le biais de variables en entrée.

Dans l'exemple illustré à la figure 12, plusieurs instances des blocs fonctionnels CTU peuvent être invoquées dans le corps du bloc fonctionnel B. On établit une référence indirecte à une instance du bloc fonctionnel CTU par une variable déclarée dans une déclaration VAR_IN_OUT à la fonction B. Dans les figures 12c) et 12d), plusieurs invocations de l'instance B1 du bloc fonctionnel B avec différents noms d'instances du bloc fonctionnel COUNTER_FB comme variables, aboutissent à des invocations de différentes instances de CTU, correspondant aux noms d'instances des variables.

Comme on le montre en 2.5.2.2, figure 11c) de la CEI 61131-3, une instance de bloc fonctionnel peut être déclarée comme une variable globale dans un programme, et le nom de l'instance de bloc fonctionnel peut alors être utilisé comme variable externe dans des blocs fonctionnels d'autres types. Bien que les instances particulières de ces blocs fonctionnels puissent changer entre les programmes, l'instance de blocs fonctionnels référencée n'est pas modifiée d'une invocation à l'autre du bloc fonctionnel qui référence (le bloc contenant la référence externe). Cette instance de blocs fonctionnels peut être utilisée, par exemple, pour fournir un moyen de fixer de l'extérieur les paramètres de toutes les instances d'un type de bloc fonctionnel qui contient le nom du bloc fonctionnel comme référence externe.

En conséquence, comme on le décrit plus haut, l'utilisation de variables externes pour les noms d'instances de blocs fonctionnels diffère de l'utilisation de variables VAR_IN_OUT pour des instances de blocs fonctionnels. Alors que l'on peut changer les variables VAR_IN_OUT d'une invocation à l'autre d'une instance de bloc fonctionnel, les références externes d'instances de blocs fonctionnels ne changeront pas, bien que les valeurs de leurs variables puissent, elles, changer.

Ce fait a des implications sur la vérification de validité fournie par l'environnement de support de programmation (PSE) pour la portée d'une instance de bloc fonctionnel. Il est fortement recommandé de vérifier la portée pendant l'édition, de façon que l'utilisateur n'ait pas besoin d'attendre le moment de l'exécution pour vérifier cette portée. Les conséquences de cette exigence sont que les noms d'instances de blocs fonctionnels utilisés comme arguments sont fixes et qu'on ne les stocke pas dans une variable. Il n'y a aussi aucun risque de concaténer cette chaîne au moment de l'exécution, ni d'accéder à un tableau de chaînes conservant le nom de cette instance de bloc fonctionnel.

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 11;

– an external variable.

Examples of this usage are shown in figures 11b) and 11c), 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 the preceding subclause, this precludes the graphical representation (which implies invocation) of indirectly referenced function block instances established via input variables.

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 figures 12c) and 12d), 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 2.5.2.2, figure 11c) of IEC 61131-3, a function block instance can be declared as a *global variable* in a *program* and the function block instance name can then be used as an *external* variable in function blocks of other types. Although the particular instances of such function blocks may vary between programs, the referenced function block instance does not vary from one invocation to the next of the *referencing* function block (the block containing the external reference). Such a function block instance may be used, for example, to provide means for externally setting the parameters of all instances of a function block type that contains the function block name as an external reference.

Thus, the use of external variables for function block instance names differs from the use of VAR_IN_OUT variables for function block instances as described above. Whilst 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 programming support environment (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.

a)
```
FUNCTION_BLOCK B
   VAR_IN_OUT COUNTER_FB: CTU; END_VAR
   VAR REACHED: BOOL; END_VAR
   ...
   (* Invocation de la variable COUNTER-FB *)
   COUNTER_FB (...);
   (* Accès à la sortie Q de la variable COUNTER_FB *)
   REACHED:= COUNTER_FB.Q;
   ...
END_FUNCTION_BLOCK
```

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

c)
```
PROGRAM P2
...
   VAR REACHED: BOOL;
    B1: B;
    COUNTER1: CTU;
    COUNTER2: CTU;
   END_VAR
   ...
   (* Accès à la valeur de COUNTER1.Q avant l'invocation *)
   REACHED:= COUNTER1.Q
   ...
   (*L'invocation de B1 provoque l'invocation de Ccounter1*)
   B1 (COUNTER_FB:= COUNTER1,...);
   ...
   (*Accès à la valeur de COUNTER1.Q après l'invocation*)
   (* L'invocation de B1 provoque l'invocation de COUNTER2 *)
   B1 (COUNTER_FB:= COUNTER2,...);
END_PROGRAM
```

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

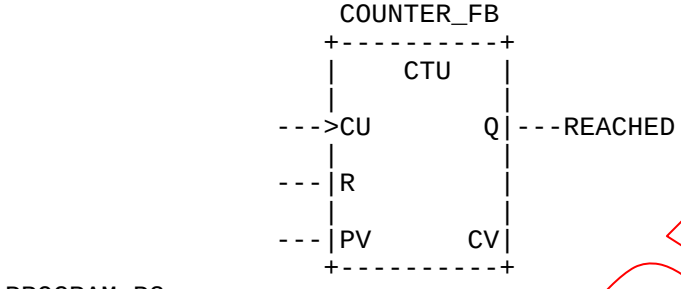
IEC 1625/99

NOTE La suppression des détails inutiles est signalée par le signe (...).

Figure 12 – Invocation d'une instance de bloc fonctionnel référencée indirectement
a) Déclaration et invocation littérale;
b) Invocation graphique;
c) Passage littéral d'un nom d'instance;
d) Passage graphique du nom d'instance

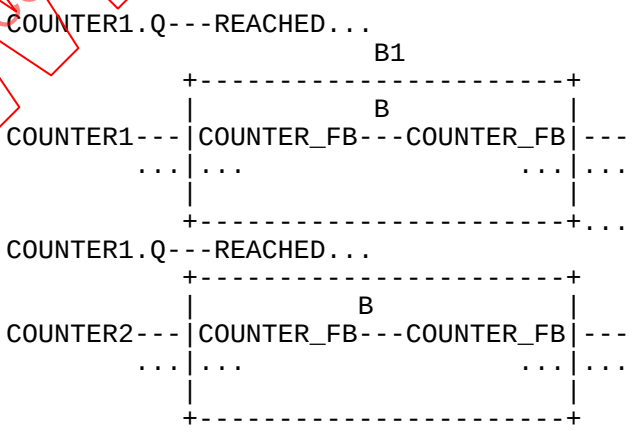
a)
```
FUNCTION_BLOCK B
   VAR_IN_OUT COUNTER_FB: CTU; END_VAR
   VAR REACHED: BOOL; END_VAR
   ...
   (* Invocation of variable COUNTER-FB *)
   COUNTER_FB (...);
   (* Access to output Q of variable COUNTER_FB *)
   REACHED := COUNTER_FB.Q;
   ...
END_FUNCTION_BLOCK
```

b)



c)
```
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 *)
   ...
   (* Invocation of  B1 causes invocation of COUNTER2 *)
   B1 (COUNTER_FB := COUNTER2, ...);
   ...
END_PROGRAM
```

d)



IEC 1625/99

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

Figure 12 – Invocation of an indirectly referenced function block instance
a) Textual declaration and invocation;
b) Graphical invocation;
c) Textual passing of instance name;
d) Graphical passing of instance name

3.5.4 Récursivité des instances de blocs fonctionnels référencées indirectement

Conformément à 2.5 de la CEI 61131-3, les invocations des POU (unités d'organisation de programmes) ne doivent pas être récursives. Cela est vrai aussi pour l'invocation de blocs fonctionnels.

Etant donné que la CEI 61131-3 définit la POU comme un type de bloc fonctionnel et non pas comme une instance de bloc fonctionnel, la vérification des invocations récursives de blocs fonctionnels peut s'effectuer en vérifiant le corps du type du bloc fonctionnel. Cela est vrai même quand le mécanisme d'utilisation des noms d'instances de blocs fonctionnels comme arguments est supporté, parce que le nom d'instance d'un bloc fonctionnel invoqué peut changer d'une invocation à l'autre, mais pas le type. Le type du bloc fonctionnel invoqué est déterminé par la déclaration dans le corps de la POU qui invoque.

La récursivité dans les langages de programmation des automates programmables est examinée plus en détail en 3.6.

3.5.5 Contrôle de l'exécution des instances de blocs fonctionnels référencées indirectement

Pour éviter les ambiguïtés dans la détermination du contrôle de l'exécution des instances de blocs fonctionnels référencées indirectement, on recommande les mesures suivantes:

1) éviter des associations directes de TASKS (tâches) à ces blocs fonctionnels, comme le décrit 2.7 de la CEI 61131-3;
2) restreindre si possible l'utilisation de ces blocs fonctionnels à des algorithmes écrits en langage ST (littéral structuré) défini en 3.3 de la CEI 61131-3, qui contient une instruction d'invocation de blocs fonctionnels explicite;
3) fournir les outils pour établir un contrôle sans ambiguïté de l'exécution, quand une application rend inévitable l'utilisation graphique de ces blocs fonctionnels.

3.5.6 Utilisation des instances de blocs fonctionnels référencées indirectement dans des fonctions

Comme le mentionne le paragraphe 2.5.2 de la CEI 61131-3, un nom d'instance de bloc fonctionnel peut être utilisé comme variable en entrée d'une fonction. Dans le corps de cette fonction, on peut lire les variables en sortie de l'instance du bloc fonctionnel indirectement référencée.

A première vue, cela semble être en contradiction avec la définition d'une fonction donnée en 2.5.1 de la CEI 61131-3. Un appel à une fonction avec les mêmes variables en entrée doit toujours générer la même valeur en sortie. Cela peut ne pas être vrai, si le même nom d'instance de bloc fonctionnel a été passé à une fonction et que cette instance de bloc fonctionnel a été entre temps invoquée. Une recherche plus détaillée de ce fait montre cependant qu'un accès en lecture à une instance de bloc fonctionnel est similaire à un accès en lecture à une variable structurée. Pour obtenir la même valeur de la fonction, il ne suffit donc pas de vérifier que le nom de l'instance du bloc fonctionnel est identique pour tous les appels, mais aussi que les valeurs de toutes les variables en sortie de l'instance du bloc fonctionnel référencée sont identiques.

3.6 Récursivité dans les langages de programmation des automates programmables

Comme il est indiqué en 2.5 de la CEI 61131-3, les appels récursifs de POU (unité d'organisation de programmes) sont interdits. Cela signifie qu'une POU ne doit pas contenir ni invoquer une autre POU du même type. Cela est vrai aussi quand la récursivité n'est pas directe mais indirecte, par exemple si une POU A appelle une POU B et que cette POU B appelle la POU A de nouveau.

3.5.4 Recursion of indirectly referenced function block instances

According to 2.5 of IEC 61131-3, invocations of program organization units (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 done 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.

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:

1) direct association of TASKS to such function blocks, as described in 2.7 of IEC 61131-3, should be avoided;
2) use of such function blocks should be restricted if possible to algorithms written in the structured text (ST) language defined in 3.3 of IEC 61131-3, which has explicit function block invocation statements;
3) where an application makes graphic use of such function blocks unavoidable, the Programming Support Environment (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 given 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.

3.6 Recursion within programmable controller programming languages

As stated in 2.5 of IEC 61131-3, recursive calls of program organization units (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 a POU A again.

Il est recommandé, à cause du temps d'exécution, de faire vérifier par le PSE pendant l'édition, l'éventuelle récursivité des programmes. Cela demande la possibilité de détecter les récursivités possibles en parcourant et en contrôlant l'arbre hiérarchique des appels statiques. Dans toute combinaison possible du trajet à partir de PROGRAMS (programmes) associés à une TASK (tâche) jusqu'au plus lointain des appels d'une POU, il ne doit pas y avoir de noms de POU qui apparaissent plus d'une seule fois.

La vérification pendant l'édition de la récursivité est aussi possible quand on utilise les noms des instances de blocs fonctionnels comme arguments, car la récursivité est basée sur les *types* de blocs fonctionnels plutôt que sur les *instances* de blocs fonctionnels.

3.7 Invocations simples ou multiples

Certains automates programmables ne permettent que l'attribution d'une seule valeur à toute entrée d'une fonction ou d'un bloc fonctionnel, alors que d'autres automates autorisent de multiples attributions de valeurs à toutes les entrées. Si on considère un système d'automates programmables comme le remplaçant d'un système de circuits électriques, il ne peut y avoir d'attributions multiples car elles introduiraient des «courts-circuits» des sorties, équivalant aux circuits non câblés des diagrammes de sorties des blocs fonctionnels. Il n'y a d'autre part pas de raison pour qu'on ne puisse pas invoquer de plusieurs endroits, avec des valeurs d'entrée différentes pour chaque invocation, un bloc fonctionnel implanté dans un système numérique. L'exemple de l'utilisation du bloc fonctionnel «SEMA» en 2.5.2.3.1 de la CEI 61131-3 illustre cette situation. L'instance de bloc fonctionnel «Printer» de type «SEMA» sera appelée deux fois, une fois pour annoncer un usage exclusif de l'imprimante, une fois pour la libérer.

Un exemple d'invocation multiple en langage littéral structuré est donné ci-après:

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

Dans cet exemple, un bloc fonctionnel nommé «aFB» est supposé avoir une variable booléenne en entrée appelée «IN». En fonction de la valeur courante d'une certaine expression booléenne, ce bloc fonctionnel est invoqué, soit avec la valeur 0 soit avec la valeur 1 pour cette entrée. Il y a deux invocations du bloc fonctionnel aFB avec différentes valeurs données au paramètre d'entrée. Dans un système qui permet de multiples assignations, cela ne pose aucun problème, mais, dans un système qui n'accepte strictement qu'une seule assignation de valeur, l'instruction IF devra être reformulée de la façon suivante:

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

En fonction de la complexité du bloc fonctionnel et du nombre d'entrées qui dépendent de valeurs calculées, l'invocation des blocs fonctionnels devient de plus en plus compliquée et cache l'objectif prévu du bloc fonctionnel. Si, dans l'exemple ci-dessus, l'entrée «IN» n'était pas du type BOOL mais du type INT, une fonction de sélection binaire «SEL» serait nécessaire.

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

Une assignation unique stricte pourrait aussi augmenter le nombre de variables intermédiaires nécessaires pour désaccoupler l'assignation de valeurs aux entrées des blocs fonctionnels.

On peut trouver des exemples similaires d'assignations multiples en SFC (diagramme fonctionnel en séquence), où une instance de bloc fonctionnel peut être invoquée par plus d'une action.

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, whereas 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 function block diagrams. 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. The example in 2.5.2.3.1 of IEC 61131-3 for the use of “SEMA” function block is an example of such a situation. The function block instance “Printer” of type “SEMA” will be called twice, once to claim an exclusive use of a printer and once to release it.

An example of multiple invocation in structured text 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 assignments this would not cause any problems, but, in a strict 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 be required.

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

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

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

La CEI 61131-3 ne stipule pas si un système utilise strictement des invocations uniques ou s'il autorise des invocations multiples. Les deux ont des avantages et des inconvénients. Avec le système d'allocation unique, il y a un endroit et un seul où l'on assigne des valeurs aux variables en entrée, ce qui augmente la fiabilité et la maintenabilité du logiciel. Avec les invocations multiples, le système d'automates programmables peut être plus «maintenable» à cause d'un niveau plus bas d'imbrication (nesting) des fonctions et il peut avoir de meilleurs temps de réponse et de possibilités de traitement en évitant les calculs superflus.

Il est des cas où il est difficile de trouver la place d'une invocation de bloc fonctionnel et il est des situations où ce qu'il faut faire et quand il faut le faire n'est pas clair. S l'on considère un cas où le bloc fonctionnel «aFB» de l'exemple précédent est connecté à une tâche cyclique, cette construction va rendre ambiguë l'invocation elle-même ainsi que les paramètres en entrée transmis et doit être considérée comme une erreur.

3.8 Dispositifs spécifiques aux langages

3.8.1 Fonctionnalités déclenchées par des fronts

Dans la CEI 61131-3, on a défini plusieurs mécanismes de fonctionnalités déclenchées par des fronts montants ou des fronts descendants:

– en tant que blocs fonctionnels, en 2.5.2.2;
– en tant que blocs fonctionnels R-TRIG et F_TRIG, en 2.5.2.3.2;
– en tant que contacts et des bobines de détection de transition positifs (P) ou négatifs (N) dans le langage LD (à contacts), en respectivement 4.2.3, tableau 61 et 4.2.4, tableau 62.

3.8.1.1 Déclenchement de fronts dans le langage LD

Comme il est noté en 2.1, beaucoup de systèmes d'automates programmables utilisent des représentations cycliques pour mettre en oeuvre des contrôles par échantillons de données. Dans la notation de la CEI 61131-3, on réalise cela en associant le programme qui met en oeuvre l'algorithme de contrôle avec une TASK (tâche) périodique décrite en 2.7.2 de la CEI 61131-3. Il convient que les utilisateurs soient avertis que, dans de tels systèmes, dans les conditions les plus défavorables, les effets d'un changement d'état dans une entrée déclenchée par un front pourraient n'apparaître dans les sorties du système que deux cycles plus tard, comme on le note dans la figure 3. Cela peut être illustré par le réseau LD simple et le diagramme temporel indiqués à la figure 13.

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 more maintainable because of a lower level of nesting of functions and may have better responsiveness and processing capacity by avoiding superfluous calculations.

There are situations where it is difficult to find the place of a function block invocation and there are situations where it is unclear what has to be done in which moment. Consider a case where function block “aFB” in the example above is connected to a cyclic task. Such a construction will make the invocation itself and the passed parameter for the input ambiguous and should be regarded as an error.

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 ladder diagram (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, 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. This can be illustrated by the simple LD network and timing diagram shown in figure 13.

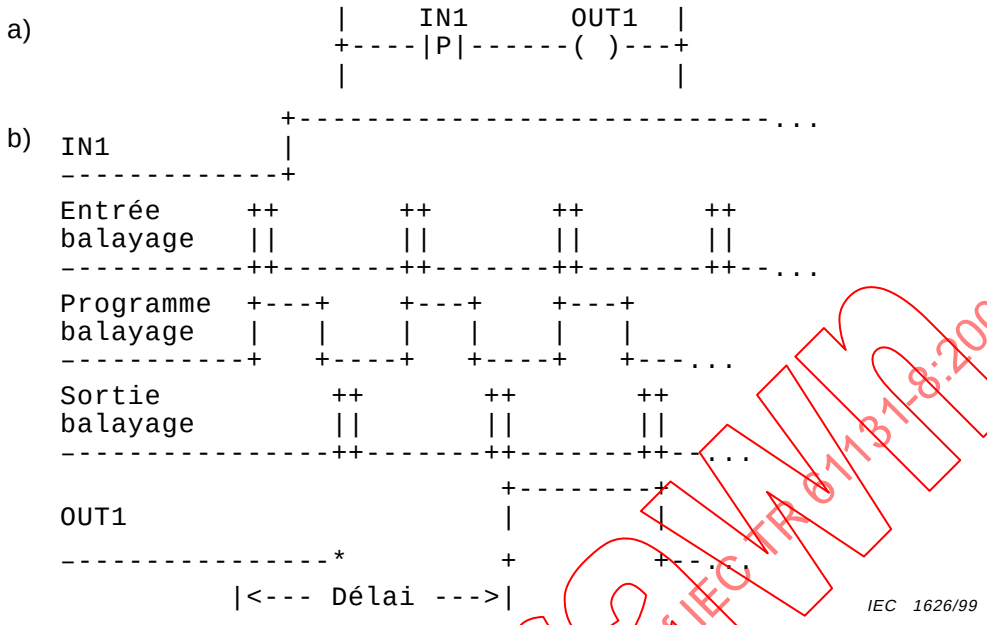


Figure 13 – Estimation des durées des fonctionnalités déclenchées par des fronts
a) Exemple de réseau
b) Estimation du cas le plus défavorable

3.8.1.2 Utilisation des blocs fonctionnels déclenchés par des fronts

Les blocs fonctionnels R_TRIG et F_TRIG définis en 2.5.2.3.2 de la CEI 61131-3 mettent en évidence différents comportements succédant à un départ à froid, comme on le décrit en 2.4.2 de cette norme. La sortie «Q» d'une instance de R_TRIG peut être mise à «1» à la première invocation, mais les sorties «Q» d'une instance de F_TRIG demande toujours au moins deux invocations avant d'être mise à «1». Ce comportement s'explique comme suit:

– comme la valeur par défaut d'une variable booléenne est «0», si l'entrée «CLK» est «1» à la première invocation, cela signifie que l'entrée est passée de la valeur «0» à la valeur «1», c'est-à-dire qu'on a détecté un front montant; la valeur de «Q» de l'instance de R-TRIG est ainsi mise à «1»;
– dans le cas d'une instance de F_TRIG, l'entrée «CLK» doit avoir la valeur «1» avant que l'on puisse détecter un changement d'état: passage de «1» à «0»; il faut donc au moins deux invocations.

De la description ci-dessus, il résulte qu'un usage intéressant de R_TRIG est un «mécanisme de détection de premier cycle». Si l'entrée «CLK» d'une instance R-TRIG est la constante «1» (ou TRUE = vrai), la sortie «Q» ne sera «vraie» que pour la première invocation car aucun passage subséquent de «0» à «1» n'est alors possible. Cela peut être utilisé comme mécanisme de détection de premier cycle, par exemple:

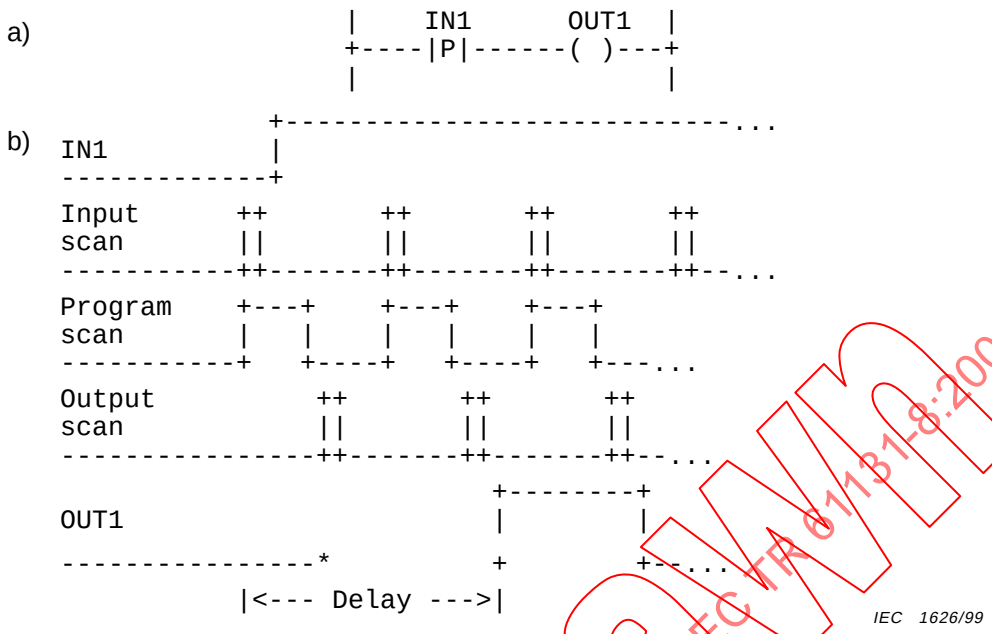


Figure 13 – Timing of edge triggered functionality
a) Example network
b) Worst-case timing

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 behaviours following “cold restart” as described in 2.4.2 of that standard. 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 “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 * premier cycle seulement *)
      ...
    ELSE
      ...
    END_IF;
...
```

3.8.2 Utilisation des blocs fonctionnels EN/ENO dans les fonctions et les blocs fonctionnels

Le paragraphe 2.5.1.2 de la CEI 61131-3 définit l'entrée «EN» et la sortie «ENO» booléennes, qui peuvent être utilisées pour contrôler l'exécution d'une fonction. Ces variables sont typiquement utilisées pour fournir un «flux d'énergie binaire» dans les fonctions du langage «à contact» (LD). Elles peuvent être utilisées également dans le langage FBD et elles sont particulièrement utiles pour lever les ambiguïtés qui peuvent être causées par l'utilisation des éléments de contrôle d'exécution graphiques décrits en 4.1.4 de la CEI 61131-3.

La figure 14 montre une application de ce principe à des parties du corps de FBD du bloc fonctionnel STACK_INT dans l'exemple donné à l'article F.4 de la CEI 61131-3. Les combinaisons des entrées booléennes PUSH, POP et R1 servent avec les variables EN/ENO à réaliser un contrôle d'exécution sans ambiguïté, sans sauts et sans étiquettes.

Conformément à la règle 3) de 2.5.1.2 de la CEI 61131-3, la sortie ENO d'une fonction doit être remise à FALSE (0) dans le cas d'une condition d'erreur lors de l'exécution de la fonction. Ce dispositif peut servir à prévenir la propagation de valeurs erronées à travers les évaluations fonctionnelles subséquentes. L'utilisation de ENO pour une signalisation et un diagnostic intensifs d'erreurs d'exécution n'est cependant pas recommandée; il vaut mieux employer à la place les procédures décrites en 4.6.2.

```
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 designs 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 ladder diagram (LD) language. However, they can also be used in the function block diagram (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.

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

According to rule 3) of 2.5.1.2 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 should be employed.

a)
```
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
```

b)

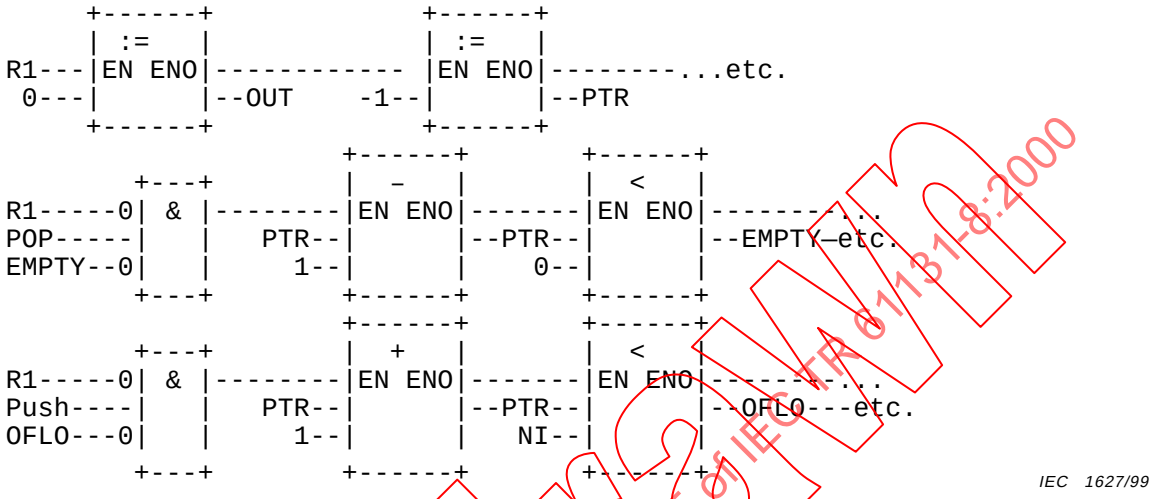


Figure 14 – Exemple de contrôle d'exécution (voir article F.4 de la CEI 61131-3)
a) Langage ST sans EN/ENO
b) Langage FBD avec EN/ENO

3.8.3 Utilisation de langages non définis dans la CEI 61131-3

Le paragraphe 1.4.3 de la CEI 61131-3 stipule que d'autres langages de programmation, tels que Pascal et C peuvent être utilisés en plus de ceux définis dans la CEI 61131-3, pour la programmation des fonctions et des blocs fonctionnels. Indépendamment du langage utilisé pour cette programmation, il faut fournir les interfaces d'informations, telles qu'elles ont été discutées en 5.5, afin de permettre l'utilisation de ces POU (unités d'organisation de programmes) dans le PSE (environnement de support de programmation).

3.9 Utilisation des éléments SFC

Les éléments du langage SFC permettent une représentation graphique claire et logique des structures séquentielles des programmes de commande et de parties de programmes de commande.

Un SFC (diagramme fonctionnel en séquence) se compose d'un ou plusieurs réseaux de STEPS (phases) et de TRANSITIONS; associé à chaque phase, un ensemble d'ACTIONS. Une action représente les opérations associées à une ou plusieurs phases; elle peut être formée par

– un ensemble d'instructions dans le langage IL;
– un ensemble de déclarations dans le langage ST;
– un ensemble de barres dans le langage LD;
– un ensemble de réseaux dans le langage FBD;
– un organigramme de fonctions séquentielles;
– une variable booléenne.

a)
```
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
```

b)

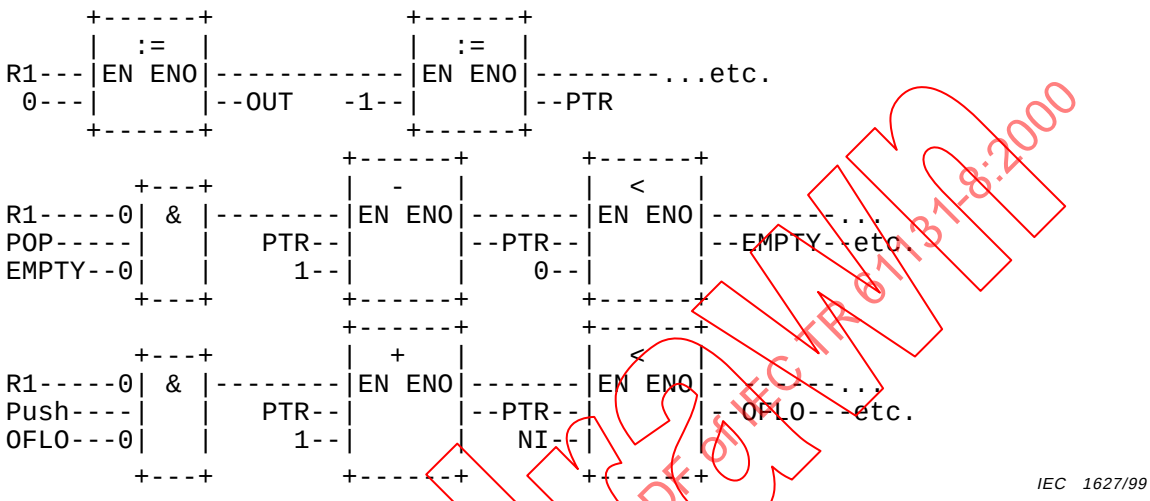


Figure 14 – Execution control example (see clause F.4 of IEC 61131-3)
a) ST language without EN/ENO
b) FBD language with EN/ENO

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 must be provided in order to permit the use of these program organization unit types in the programming support environment (PSE).

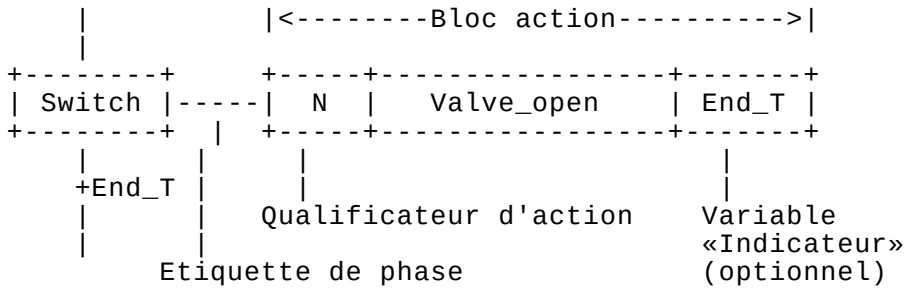
3.9 Use of SFC elements

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

A sequential function chart (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;
– a Boolean variable.

3.9.1 Commande des actions



IEC 1628/99

Figure 15 – Exemple de bloc action

L'exécution d'une action est conditionnée par l'état des phases qui lui sont associées combiné aux effets de leurs qualificateurs d'action.

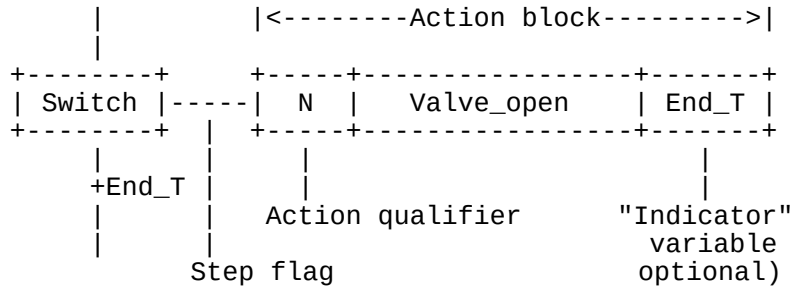
Comme il est défini en 2.6.4.5 de la CEI 61131-3, le bloc fonctionnel de commande de l'action contient une description logique des effets et des interactions des qualificateurs de l'action. Afin d'arriver à une meilleure compréhension des commandes des actions, une copie (instance) du bloc de commande de l'action est affecté à chaque action. Le bloc de commande de l'action est donc une partie constitutive d'une action: il définit l'exécution de l'action (on peut dire aussi qu'il «contrôle» l'exécution).

De cette façon, l'exécution d'une action peut être activée pour la durée d'une phase, exécutée juste une fois («impulsion»), activée pendant une période infinie («actionnée» ou «stockée»), désactivée («arrêtée»), activée avec un délai, ou activée pour une période limitée.

La CEI 61131-3 prescrit que les fonctionnalités d'un bloc de commande d'action, mais pas forcément le bloc lui-même, soient mises en oeuvre dans un système conforme au SFC d'automates programmables. On peut décrire ces fonctionnalités de façon informelle à l'aide des règles suivantes.

1) Le comportement (fonction) d'une action dans les parties du programme est principalement déterminé par les dispositifs suivants:
   - état de l'étiquette de phase
   - qualificateur de l'action
   - action
   - bloc de commande de l'action
2) Le bloc de commande de l'action décrit (logiquement) la méthode d'exploitation (fonction) et les interactions des qualificateurs d'action. (Une action peut être définie avec plusieurs qualificateurs d'action différents, dans un programme ou dans une partie du programme.) Les qualificateurs d'action sont
   - N non stockée
   - S actionnée/stockée
   - R arrêt prioritaire
   - P impulsion
   - L limitée dans le temps
   - D différée
   - DS différée et stockée
   - SD stockée et différée
   - SL stockée et limitée dans le temps

3.9.1 Action control



IEC 1628/99

Figure 15 – Example of an action block

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 function 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 function block is assigned to each action. Therefore, the action control function block is a constituent part of an action: it defines the execution of the action (one could also say that it “controls” the execution).

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

1) 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 function block
2) The action control function 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 time delayed and stored

   SD stored and time delayed

   SL stored and time limited

La figure 15 en 2.6.4.5 de la CEI 61131-3 représente le flux logique des activations comme un diagramme de câblage. On peut voir que l'activation d'une phase, conformément aux attributs choisis, s'effectue via une connexion avec un bloc de commande d'action. La sortie Q du bloc de commande d'action est assignée soit directement à une variable booléenne, soit sert à l'appel de l'action réelle. Dans ce dernier cas, il faut tenir compte du fait que l'activation de l'action prend place soit pour Q = 1, soit pour le front descendant de Q.

Toute section d'un réseau SFC comprend habituellement une succession de blocs action, qui diffèrent souvent uniquement sur le choix des attributs. Plusieurs actions différentes peuvent en outre être connectées à une seule phase, comme le montre la figure 16a) en 2.6.4.5 de la CEI 61131-3. On peut donc faire les remarques générales suivantes:

– chaque phase peut être connectée à plusieurs blocs action;

– chaque bloc action est cependant connecté à une phase;

– chaque action est connectée à un seul bloc de commande d'action;

– chaque bloc de commande d'action est connecté à un ou plusieurs blocs action.

La figure 16b) de la CEI 61131-3 montre la transformation du réseau SFC de la figure 16a) en un bloc de diagramme de câblage, correspondant à la réalisation des blocs action du SFC comme un réseau (optimisé) de câblage. Le code réellement exécutable généré n'a pas besoin d'être identique à cette figure, pour autant que son comportement est fonctionnellement équivalent.

3) Il ne peut se produire qu'une seule activation pendant la période dépendant d'un qualificateur (D, L, SD, DS, SL) pour un bloc action.

4) L'activation d'un qualificateur SL dans un bloc action ne permet pas le traitement d'un qualificateur SD dans le même bloc et vice versa.

Les diagrammes temporels des caractéristiques fonctionnelles des différents qualificateurs sont donnés dans le paragraphe suivant.

3.9.2 Actions booléennes

Une action qui n'agit que sur une variable booléenne est appelée *action booléenne*. Dans ce cas particulier, le comportement d'une action est défini par une assignation directe de l'état de la sortie Q de la variable booléenne.

Les figures 16a) à 16h) montrent les fonctions des qualificateurs pour les variables booléennes, basées sur la description des qualificateurs d'action de la CEI 60848. Dans ces figures, on suppose que «test» a été déclaré comme une variable de type BOOL.

NOTE Si on a besoin d'une combinaison de différents qualificateurs, il est recommandé de se référer à la description détaillée du bloc de commande de l'action en 2.6.4.5 de la CEI 61131-3 afin d'arriver à une meilleure compréhension des opérations qui en résultent.

Figure 15 of 2.6.4.5 of IEC 61131-3 represents the logical flow of the activation as a wiring diagram. Here, one can see that the activation of a step according to the chosen attribute is done through a connection within the action control block. The output Q of the action control block is assigned either to a Boolean variable directly or serves the calling up of the real action. In the latter case, it has to be taken into account that the activation of the action takes place either at Q = 1 or at the falling edge of Q.

Any section of a 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 2.6.4.5 of IEC 61131-3. Therefore, the following general observations can be made:

– each step can activate several action blocks;
– however, each action block is allocated to one step;
– each action is connected with exactly one action control block;
– each action control block is activated by 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 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.

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

4) 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.

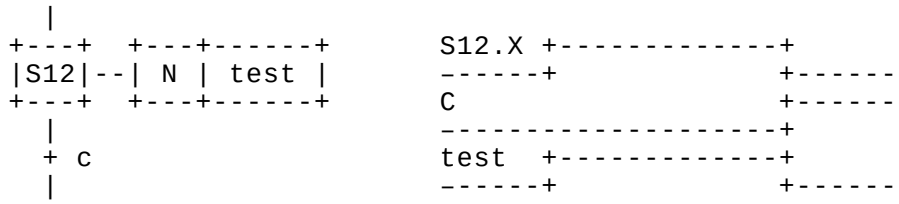
3.9.2 Boolean actions

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.

Figures 16a) through 16h) show 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.

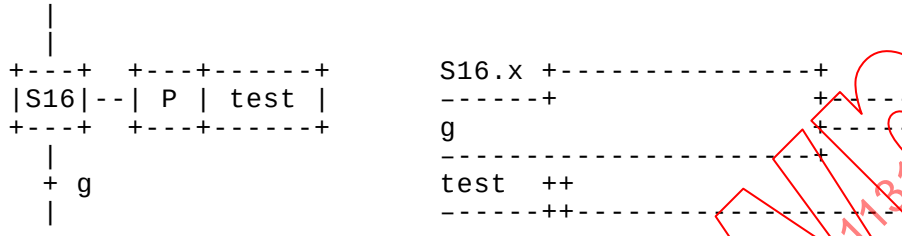
NOTE If a combination of various qualifiers is required, it is recommended to refer to the detailed description of the action control block in 2.6.4.5 of IEC 61131-3 in order to gain a better understanding of the resulting operations.

a)



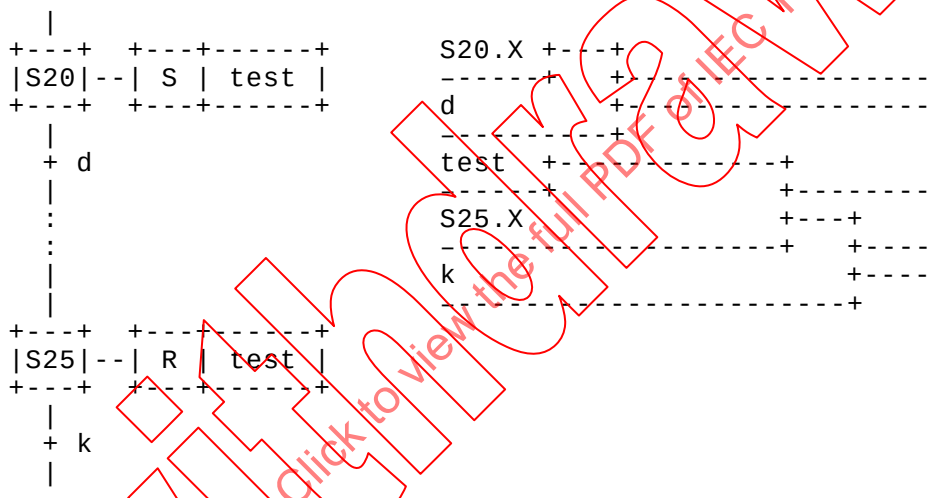
La variable booléenne «test» a une valeur aussi longtemps que S12 est actif.

b)



Dès que l'étape est active, la variable booléenne reçoit une valeur pour un cycle opérationnel.

c)

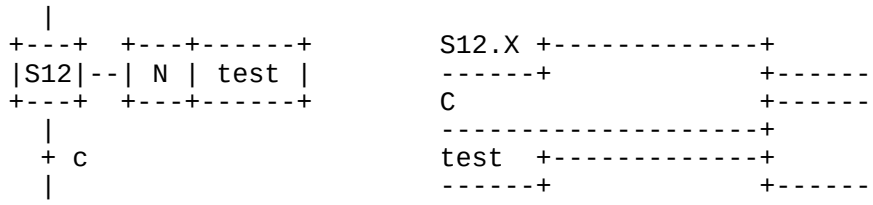


La variable booléenne reçoit une valeur et est stockée dès que l'étape est active. Elle ne peut être ré-initialisée qu'au moyen du qualificateur R (R-qualifier).

IEC 1629/99

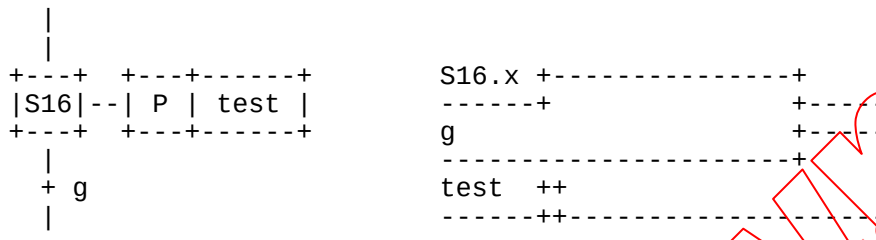
Figure 16 – Chronométrage des actions booléennes
a) Action N (non stockée)
b) Action P (pulsée)
c) Action S/R (actionnée/stoppée)

a)



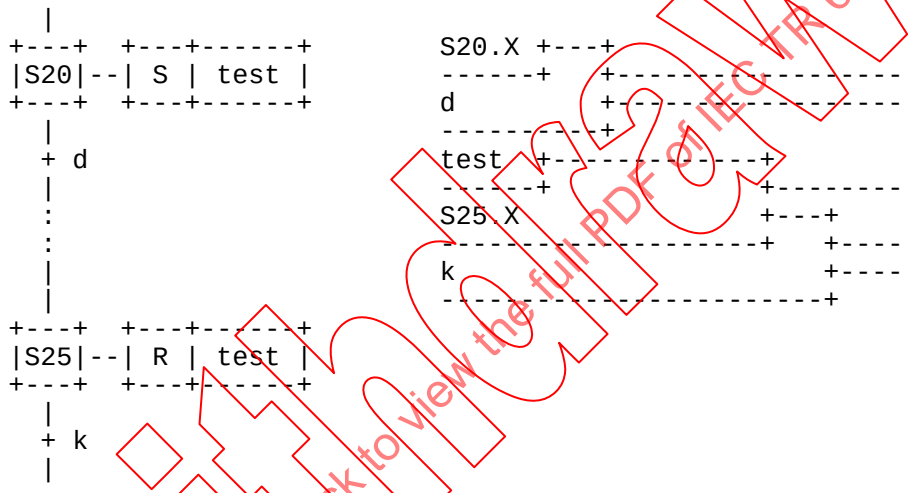
The Boolean variable "test" is set as long as step S12 is active.

b)



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

c)

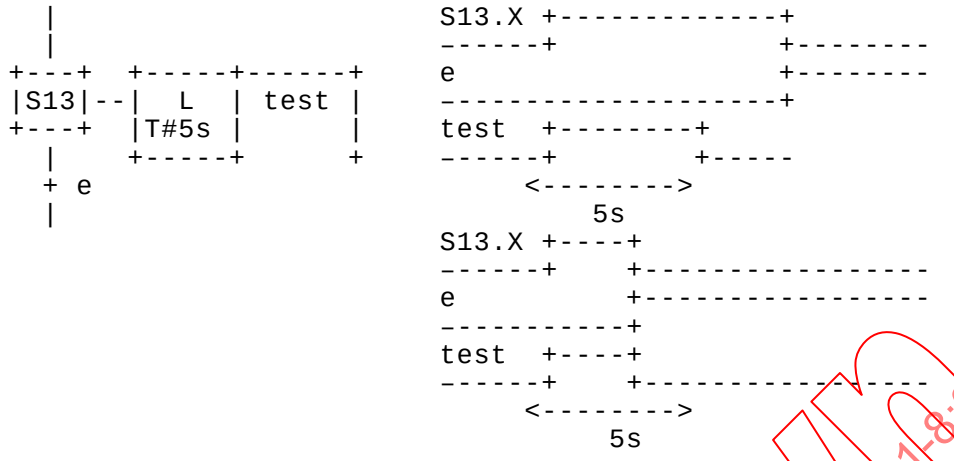


The Boolean variable is set and stored as soon as the step is active. It can only be reset by means of the R-qualifier.

IEC 1629/99

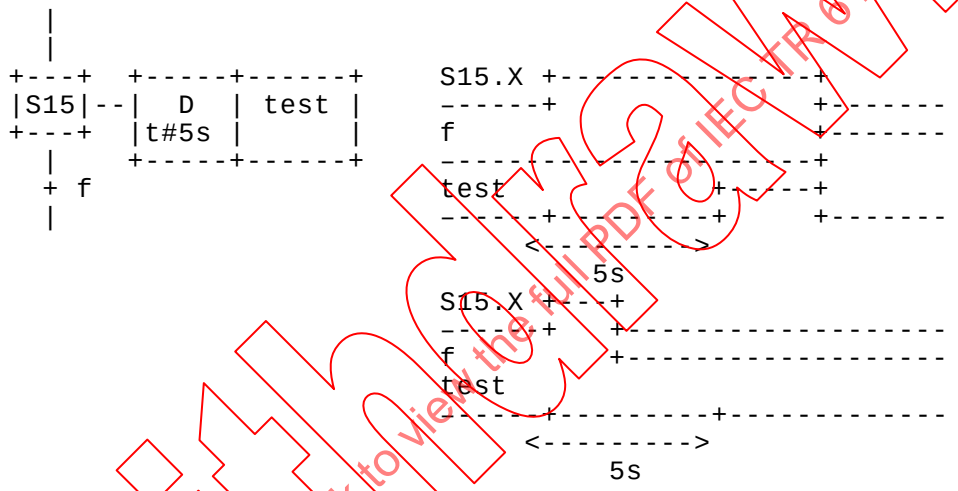
Figure 16 – Timing of Boolean actions
a) N (non-stored) action
b) P (pulse) action
c) S/R (set/reset) action

d)



La variable booléenne reçoit une valeur pour une durée prédéterminée, aussi longtemps que l'étape correspondante est active.

e)

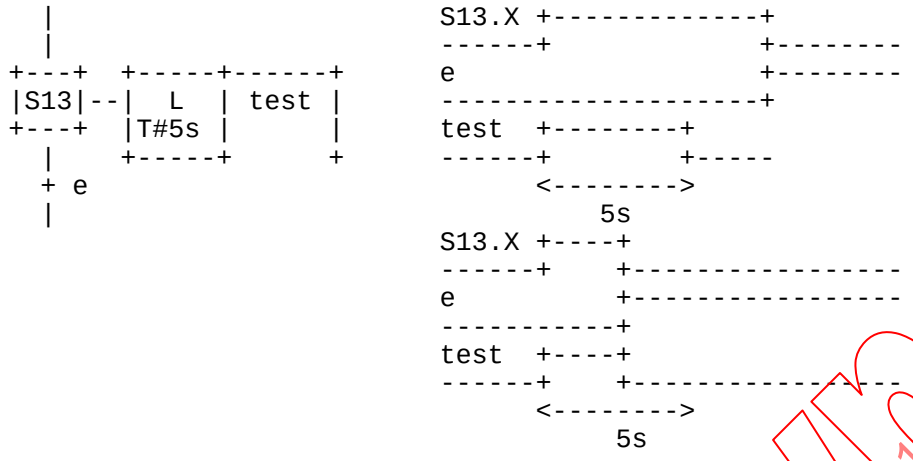


La variable booléenne reçoit une valeur après un délai prédéterminé et la garde aussi longtemps que l'étape correspondante est active.

IEC 1630/99

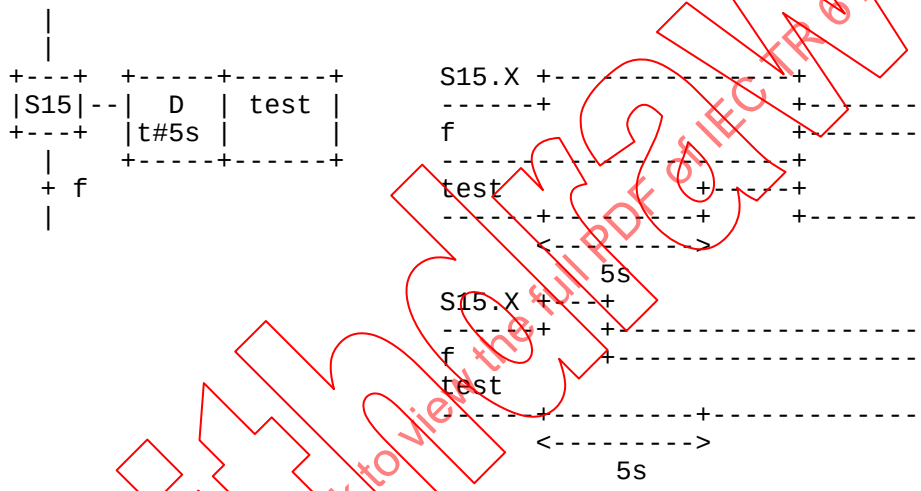
Figure 16 – Chronométrage des actions booléennes *(suite)*
d) Action L (limitée dans le temps)
e) Action D (différée)

d)



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

e)

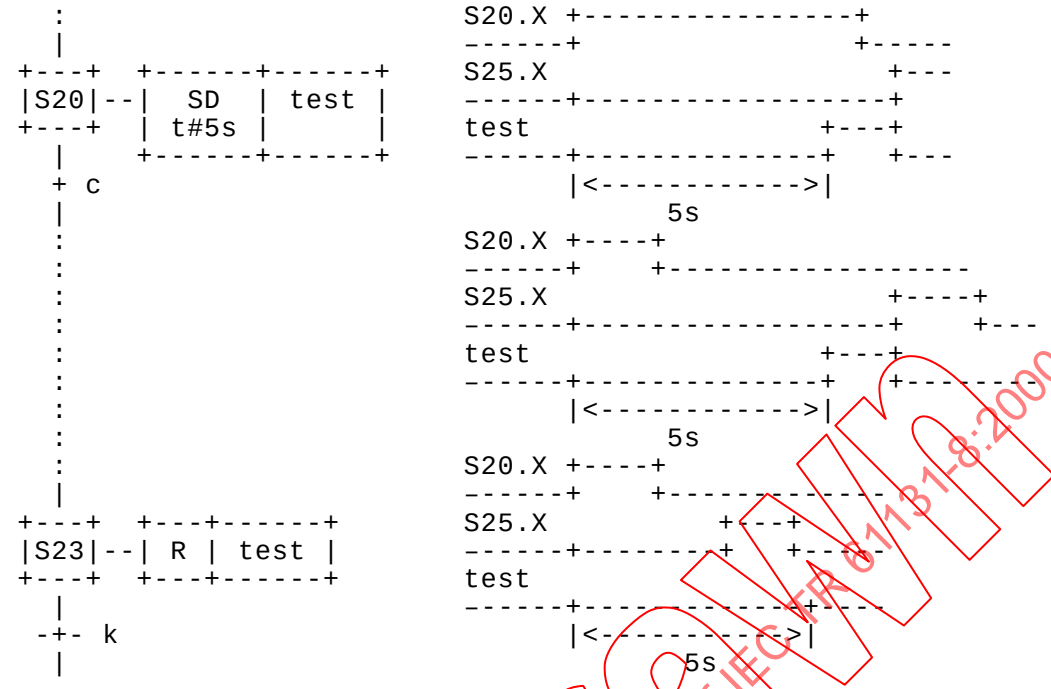


The Boolean variable is set after a preset time has elapsed and remains set for as long as the corresponding step is active.

IEC 1630/99

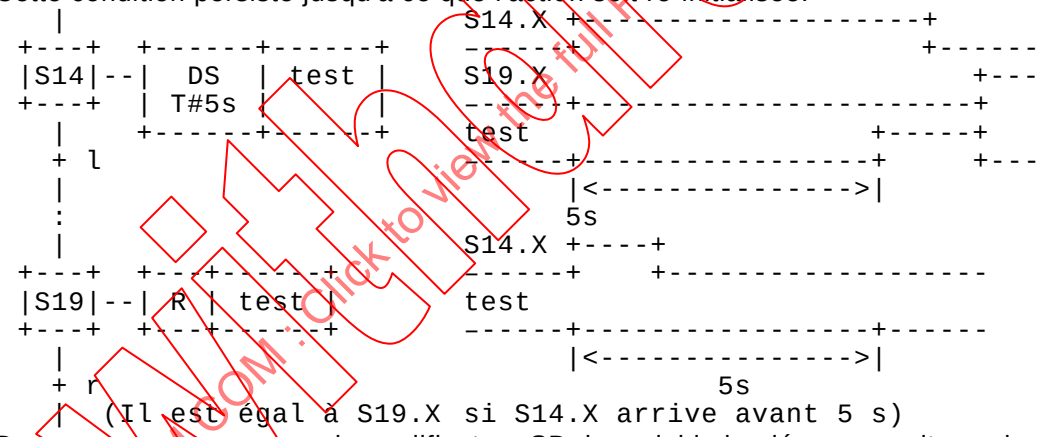
Figure 16 – Timing of Boolean actions *(continued)*
d) L (time limited) action
e) D (time delayed) action

f)



L'action est stockée et la variable booléenne reçoit une valeur après un délai prédéterminé après l'activation de l'étape, même si l'étape devient inactive. Cette condition persiste jusqu'à ce que l'action soit ré-initialisée.

g)

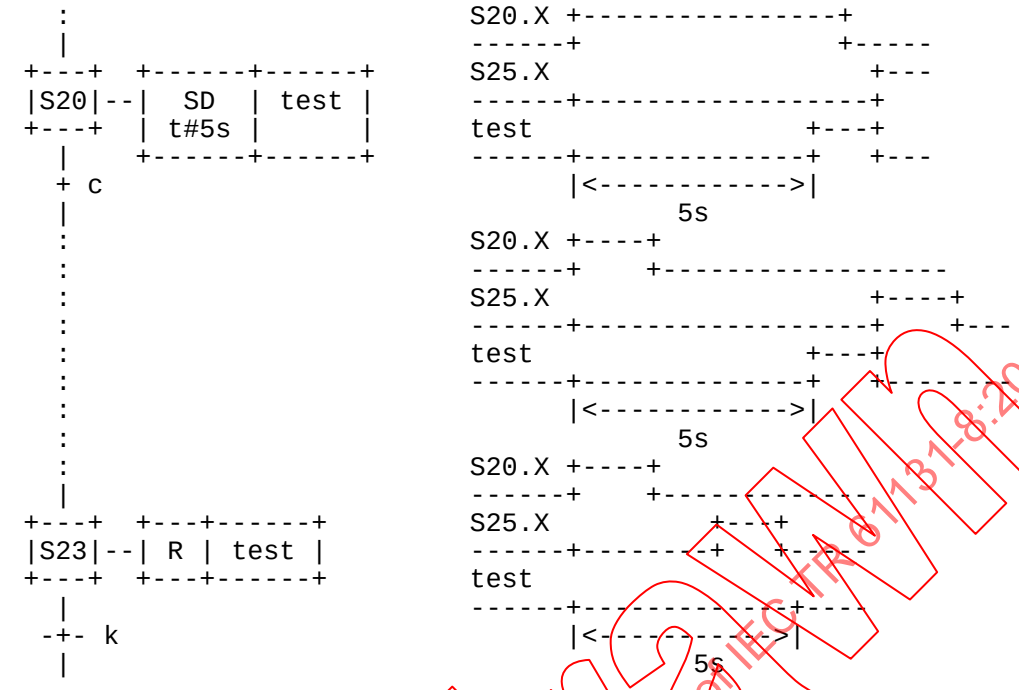


Dans ce cas, comme pour le qualificateur SD, la variable booléenne reçoit aussi une valeur après un délai, il diffère du qualificateur SD parce que l'étape correspondante doit rester active pendant le délai.

IEC 1631/99

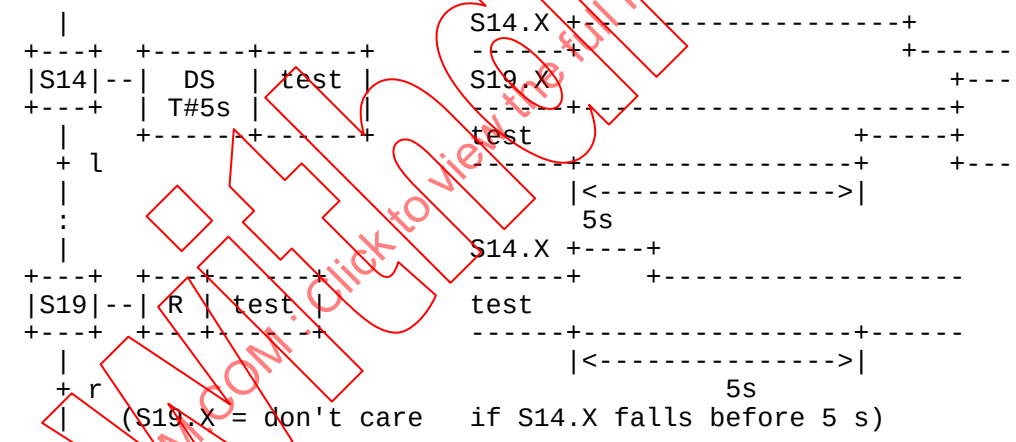
Figure 16 – Chronométrage des actions booléennes *(suite)*
f) Action SD (stockée et différée)
g) Action DS (différée et stockée)

f)



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.

g)

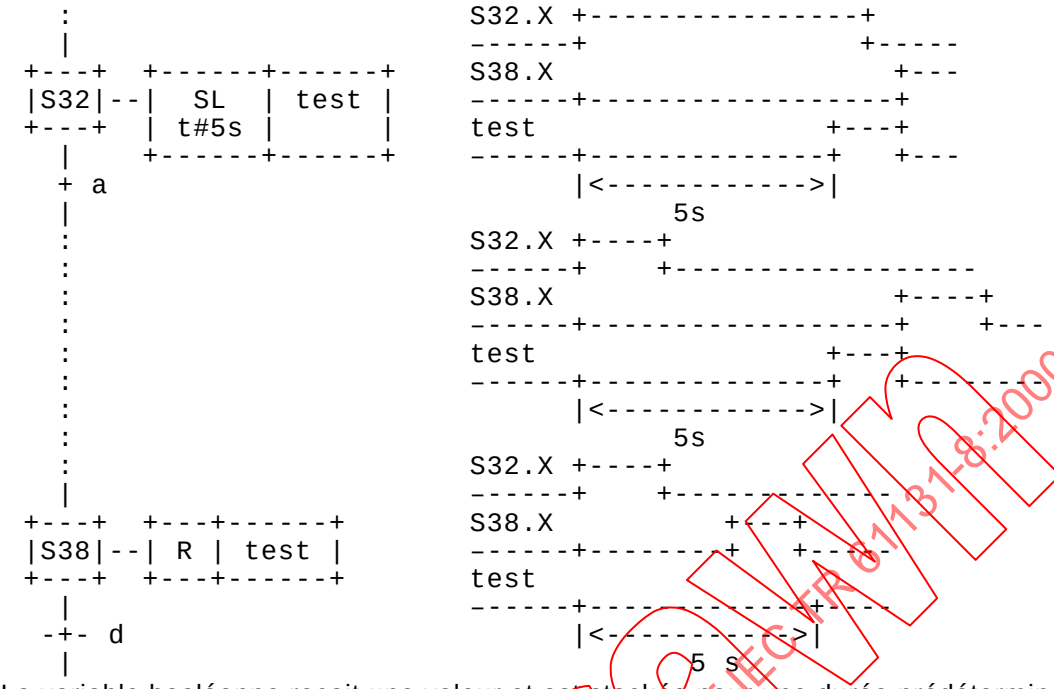


In this case, as for the SD qualifier, the Boolean variable is also set with a time delay. It differs from the SD qualifier in that the corresponding step must remain active during the time delay.

IEC 1631/99

Figure 16 – Timing of Boolean actions *(continued)*
f) SD (stored and time delayed)
g) DS (time delayed and stored)

h)

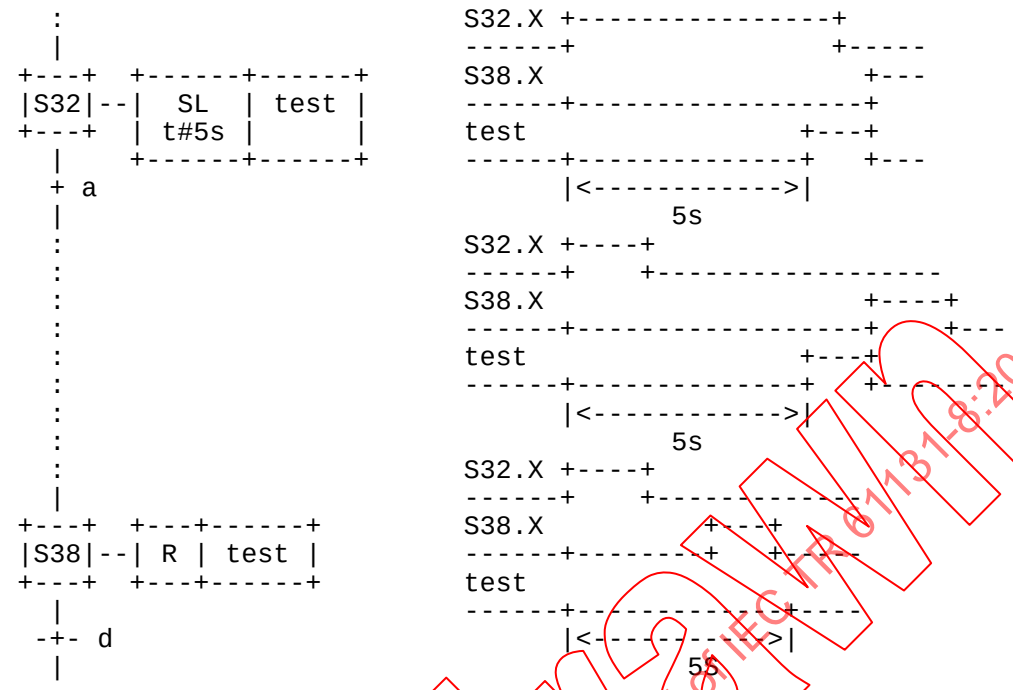


La variable booléenne reçoit une valeur et est stockée pour une durée prédéterminée dès que l'étape correspondante est active.

IEC 1632/99

Figure 16 – Chronométrage des actions booléennes *(suite)*
h) SL (stockée et limitée dans le temps)

h)



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

IEC 1632/99

Figure 16 – Timing of Boolean actions *(continued)*
h) SL (stored and time limited) action

3.9.3 Actions non-SFC

On peut, dans les actions, en plus des variables booléennes, programmer des algorithmes plus complexes, comme le montre la figure 17.

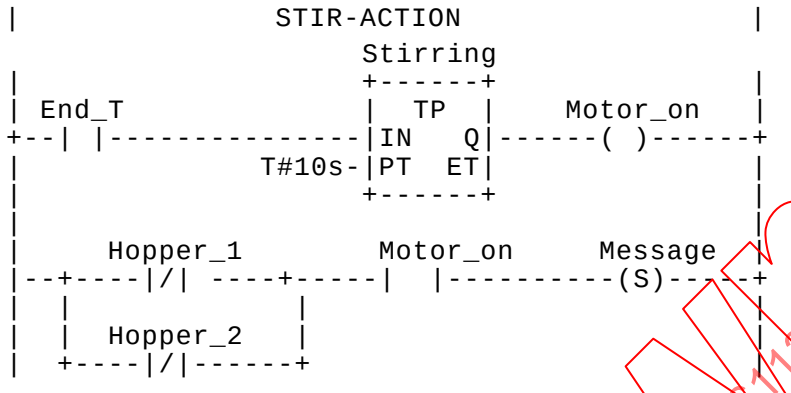


Figure 17 – Exemple d'action programmée non booléenne

Le traitement de ces actions dépend de l'état de la sortie «Q» du bloc de commande d'action correspondant. Quand la sortie «Q» est TRUE (vrai) le code de l'action est constamment validé pour être exécuté, sous le contrôle de la POU (unité d'organisation de programmes) telle que programme ou bloc fonctionnel, dans laquelle il est inclus. Quand «Q» passe de TRUE vers FALSE (faux), le code est exécuté une dernière fois. Pendant l'exécution finale, le drapeau de l'étape correspondante est déjà mis à 0.

Pendant l'élaboration de la CEI 61131-3, on a formulé les spécifications ci-dessus pour la programmation des actions afin de maximiser le contrôle de l'utilisateur sur les points suivants:

– maintien de la consistance des données calculées en garantissant que les actions programmées ne pourront pas être interrompues prématurément;

– garantie que le bit en sortie peut être forcé à la valeur correcte, par exemple en évaluant un drapeau de phase quand l'action est terminée.

Afin d'éviter des résultats inattendus, les programmeurs doivent garder en mémoire le fait que chaque action est évaluée au moins deux fois (une fois avec Q = 1 et une fois avec Q = 0). La figure 18 illustre ce point dans le contexte du qualificateur P (pulsion). Dans cet exemple, l'objectif est de maintenir un compte en nombres entiers S15_CT du nombre de fois où l'étape S15 a été activée. Comme le montre la figure 18a) ceci est réalisé par l'action A15. La figure 18b) montre un corps incorrect pour cette action qui va aboutir à ce que S15_CT soit égal à deux fois le nombre d'activations, parce que A15 sera invoquée une fois lors de l'activation de S15 et à nouveau dans le balayage suivant de S15. Les figures 18c) et 18d) illustrent les solutions possibles à ce problème, respectivement dans les langages ST et FBD.

3.9.3 Non-SFC actions

In addition to Boolean variables, more complex algorithms can also be programmed within action, as illustrated in figure 17.

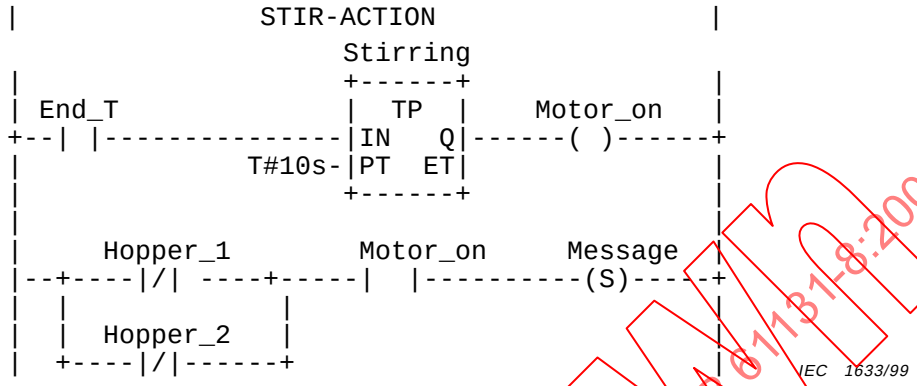


Figure 17 – Example of a programmed non-Boolean action

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, e.g. 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 18 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 18a), this is to be accomplished by action A15. Figure 18b) 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 of S15. Figures 18c) and 18d) illustrate possible solutions to this problem in the ST and FBD languages, respectively.

a)
```
    |
+-----+   +---+-----+
| S15 |---| P | A15 |
+-----+   +---+-----+
    |
```

b)
```
S15_CT:= S15_CT + 1;
```

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

d)
```
         S15_CTR
        +-----+
        | CTU |
S15.x--->CU Q|---
... ---|R    |
... ---|PV CV|---S15_CT
        +-----+
```

IEC 1634/99

Figure 18 – Utilisation du qualificateur P (pulsion)
a) Fragment en SFC;
b) Corps incorrect pour l'action A15 (langage ST)
c) Corps correct pour l'action A15 (langage ST)
d) Corps correct pour l'action A15 (langage FBD)

3.9.4 Actions SFC

Le traitement d'une action programmée en SFC n'est pas différent du traitement des actions programmées en LD, FBD, etc. Il convient cependant que l'utilisateur soit averti que seules seront exécutées les parties de SFC correspondant à des phases et à des actions actives, et les transitons activées seront exécutées selon une mise oeuvre typique de SFC. En particulier, pendant la première exécution après l'initialisation du système, seuls seront traités les éléments associés à la ou aux phases initiales.

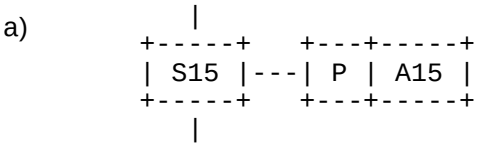
L'utilisateur doit définir toutes les interactions entre les éléments d'une action et d'autres éléments, typiquement en couplant les données en entrée et en sortie avec celles des éléments contenus dans l'action.

Une *action SFC* est une action qui contient un ou plusieurs réseaux SFC complets. Tout comme les autres actions, une action SFC peut être associée à plusieurs phases (voir 2.6.4 de la CEI 61131-3).

NOTE L'usage des actions SFC n'est PAS RECOMMANDÉ à cause des possibilités d'exécution indépendante d'actions «SFC filles» même quand l'action «parent» a été suspendue. La suppression de cette possibilité de la CEI 61131-3 est à l'étude.

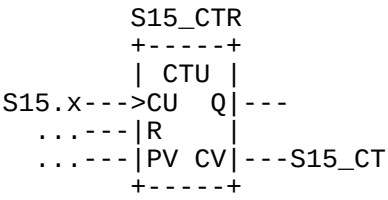
Le SFC contenu dans une action SFC est défini de la même façon (littérale ou graphique) et avec les mêmes restrictions que les SFC ordinaires. En particulier, chaque réseau SFC d'une action SFC doit contenir une phase initiale.

L'exécution d'une action SFC est contrôlée de la même façon que celle des autres actions, c'est-à-dire, par l'équivalent fonctionnel de la sortie «Q» du bloc de commande d'action correspondant. Tant que la sortie «Q» reste un «1» booléen, l'action SFC est exécutée en continu, balayage après balayage (voir point 1) de 2.6.4.5 de la CEI 61131-3). Le bloc de commande d'action est influencé par les qualificateurs d'action correspondants; on peut utiliser le jeu complet d'attributs des qualificateurs d'actions dans les actions SFC. Par un usage approprié des qualificateurs d'actions, un concepteur expérimenté peut obtenir un degré élevé de flexibilité en concevant les algorithmes de contrôle séquentiel.

a)


b)
```
S15_CT := S15_CT + 1 ;
```

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

d)


IEC 1634/99

Figure 18 – Use of pulse (P) qualifier
a) SFC fragment
b) Incorrect body for action A15 (ST language)
c) Correct body for action A15 (ST language)
d) Correct body for action A15 (FBD language)

3.9.4 SFC actions

The processing of actions programmed in SFC does not differ from the processing of actions programmed in LD, FBD, etc. However, the user should be aware that only those portions of the SFC corresponding to active steps and actions and enabled transitions will be executed in a typical SFC implementation. In particular, during the first execution after system initialization only those elements associated to the initial step(s) will be processed.

All interactions between the elements of an action and other elements must be established by the user, typically by coupling of data into and out of the elements contained in the action.

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 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 under consideration.

The SFC contained in an SFC action 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 action must contain an initial step.

The execution of SFC actions is controlled in the same way as other actions, i.e. by the functional equivalent of the "Q" output of a corresponding action control block. As long as the "Q" output stands at Boolean 1, the SFC action is executed continuously scan by scan (see item 1) of 2.6.4.5 of IEC 61131-3). The action control is influenced by the corresponding action qualifiers; the complete set of possible action qualifier attributes can be used with SFC actions. By appropriate use of action qualifiers, an experienced designer can obtain a high degree of flexibility in devising sequential control algorithms.

L'utilisation des actions SFC est différent de l'appel d'un sous-programme où on exécute un morceau de code jusqu'à ce qu'il soit terminé et où le programme continue ensuite après l'appel. Le contrôle reste au réseau SFC qui active et l'action SFC associée est exécutée au moins une fois, mais il n'y a pas d'attente de la structure SFC appelante pour la fin de l'action SFC. Une transition peut donc être effectuée immédiatement après un balayage de l'action SFC si, comme décrit en 2.6.4.3 de la CEI 61131-3, l'utilisateur n'a pas explicitement vérifié une variable «indicateur» signalant que l'action SFC est terminée.

La figure 19 décrit cette situation; la variable «indicateur» DONE contrôle la transition succédant à l'étape S1. La représentation dans la figure 19 suit le dispositif n° 1 du tableau 43 de 2.6.4.2 de la CEI 61131-3.

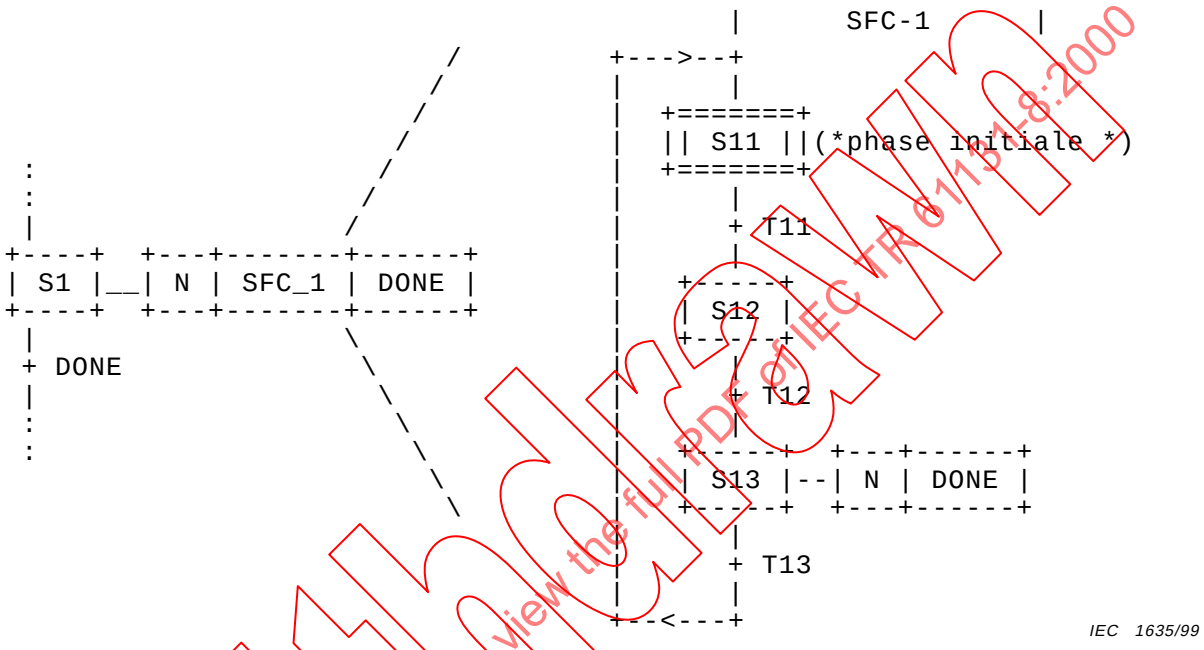


Figure 19 – Exemple d'action SFC activée par SFC

Il n'y a pas de terminaison définie pour l'évolution d'un SFC. Cela est différent d'un programme utilisant un langage qui peut être parcouru dans un ordre défini du début à la fin. On a donc besoin d'une définition pour la signification d'un balayage d'un SFC, que ce SFC constitue le sommet d'une POU ou une action SFC. Un des algorithmes possibles pour le balayage d'un SFC est:

1) détermination du jeu de phases actuellement actives; ce sont les étapes initiales pour le premier balayage suivant l'initialisation du système. Autrement, détermination de ce jeu en désactivant les étapes précédentes et en activant les étapes suivantes, mise à jour de la liste des transitions;
2) détermination de l'état des sorties «Q» de tous les blocs de commande d'actions et balayage final de toute action associée à un front descendant «Q»;
3) balayage de toutes les actions dont la valeur de «Q» du bloc de commande est «1» booléen;
4) détermination des transitions éventuelles à faire avant le prochain balayage.

On peut, à l'aide de cet algorithme de balayage, associer une action SFC à une ou plusieurs phases à travers des blocs actions de la même façon que les autres actions. S'il y a plusieurs phases actives associées à la même action SFC, toutes ces associations résultent en entrées dans le bloc de commande de cette action.

The usage of SFC actions is different from calling a subroutine where a piece of code is executed until its end and the program afterwards continues after the call location. Control stays with the activating SFC network and the associated SFC action is executed at least once, but there is no wait of the calling SFC structure for the termination of the SFC action. Thus, a transition may be cleared immediately after executing one scan of an SFC action if the user does not explicitly check for an “indicator” variable, as described in 2.6.4.3 of IEC 61131-3, indicating completion of the SFC action.

This situation is shown in figure 19, where the “indicator” variable DONE controls the clearing of the transition succeeding step S1. The representation in figure 19 follows feature No 1 in table 43 of 2.6.4.2 of IEC 61131-3.

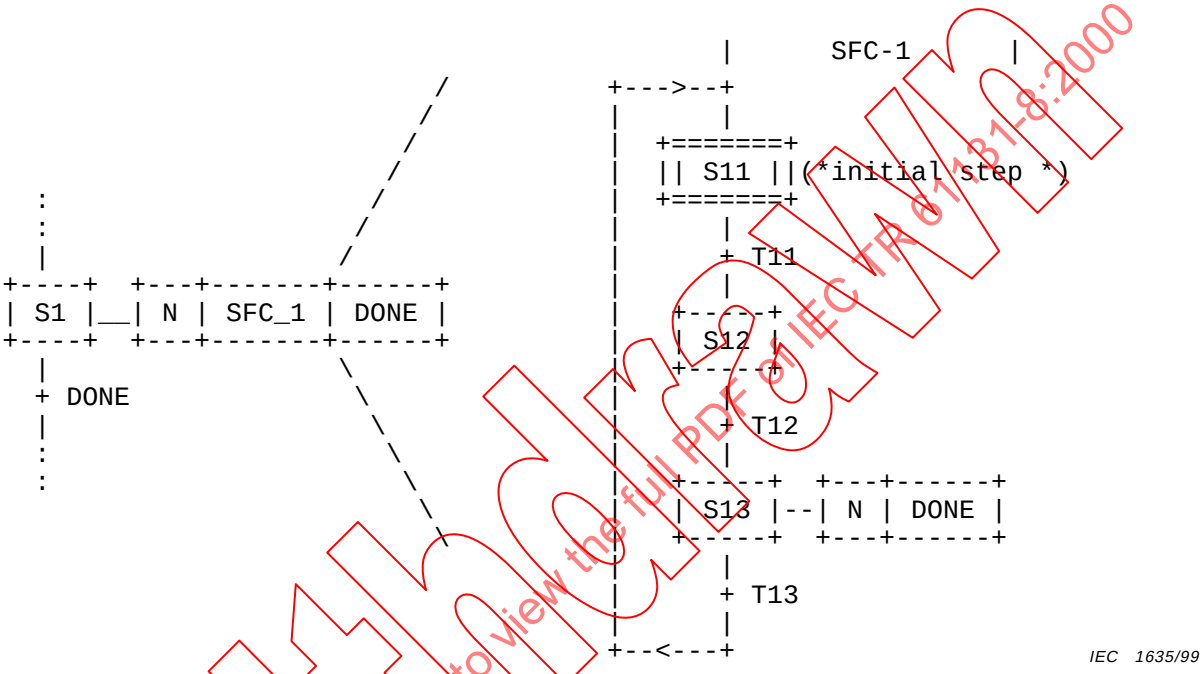


Figure 19 – Example of an SFC action enabled by an SFC

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 a start to an 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:

1) 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;
2) determine the state of all action control block “Q” outputs and perform the final scan of any actions associated with a falling edge “Q”;
3) scan all actions with action control block “Q” values of Boolean 1;
4) determine the transitions to be cleared (if any) on the next scan.

With this scan algorithm, an SFC action can be associated with one or more steps through action blocks in the same way as any other action. If there are several activated steps associated with the same SFC action, all these associations result in inputs to the action control block of this action.

Cette définition de balayage reconnaît le fait qu'il n'y a pas de boucles implicites vers l'étape initiale, quand l'étape finale est atteinte. Toutes les liaisons doivent être explicitement programmées, par exemple la liaison à travers la condition de transition T13 dans la figure 19. De cette façon, il n'y a pas de risques de perte du jeton d'un réseau SFC car toutes les transitions doivent être suivies par une phase.

A la fin du balayage, l'état de l'évolution d'un SFC est déterminé exactement de la même façon que celui de tout autre SFC, c'est-à-dire par les états de ses actions actives et (éventuellement) qui lui sont internes. L'exécution d'une action SFC n'implique donc pas un parcours complet du programme jusqu'à une fin définie; elle signifie plutôt que l'on prépare l'action appropriée pour le prochain balayage.

NOTE Il convient de prendre des précautions pour ne pas inclure dans une action SFC des éléments d'activation ou de détection de front qui pourraient ne pas fonctionner correctement si l'événement se produit alors que l'action n'est pas activée. Les actions différées (D), les minuteurs, etc. sont des exemples de tels éléments.

Bien qu'une action SFC ne soit pas une POU, il convient de ne pas utiliser d'actions SFC récursives car cela pourrait provoquer une imbrication infinie de balayages SFC.

3.9.5 Blocs fonctionnels SFC

Il n'y a pas que des actions que l'on peut définir en SFC, on peut aussi définir des blocs fonctionnels, qui seront dénommés *blocs fonctionnels SFC*. L'utilisation de ces blocs fonctionnels suit les règles communes à tous les blocs fonctionnels. L'utilisateur peut déclarer plusieurs instances du même type de bloc fonctionnel, chacun avec sa propre zone de données privées, zone qui contient aussi des informations concernant l'état en cours du SFC. Chaque instance peut alors travailler indépendamment, en utilisant le même algorithme SFC, mais avec des variables d'état internes séparées et cachées.

La figure 20 montre la déclaration d'un bloc fonctionnel SFC simple. Il faut remarquer que le SFC contenu dans un bloc fonctionnel SFC est défini de la même façon (littérale ou graphique), avec les mêmes restrictions que les SFC ordinaires. Chaque réseau SFC d'un bloc fonctionnel SFC doit, en particulier, contenir une phase initiale.

On peut invoquer les instances de blocs fonctionnels SFC de la même manière que tous les autres blocs fonctionnels. Les variables en entrée qui commandent l'évolution du SFC peuvent être déclarées et utilisées. On peut, par exemple, passer une condition de transition booléenne via une variable en entrée, pour déparer une évolution, comme T11 dans la figure 20. Une action déclarée localement dans le corps d'une fonction SFC, peut aussi accéder aux variables du bloc fonctionnel.

L'exécution d'une instance de bloc fonctionnel SFC est invoquée de la même façon que tout autre bloc fonctionnel, elle dépend des mécanismes d'invocation disponibles dans le langage utilisé pour programmer l'instance du bloc fonctionnel. Le corps du bloc fonctionnel est ensuite exécuté à l'aide d'un algorithme de balayage SC tel que celui qui est esquissé en 3.9.4.

A cause des restrictions provoquées par le fait que la récursivité des POU est interdite, un bloc fonctionnel SFC ne doit pas contenir d'invocation d'une instance de bloc fonctionnel de même type.

3.9.6 Variables «indicateur»

Comme il est indiqué en 2.6.4.3 de la CEI 61131-3, on peut spécifier une variable «indicateur» dans un bloc action afin d'indiquer si l'exécution de l'action a été ou non correcte et de transmettre cette information à des fins de traitements ultérieurs, par exemple en tant qu'information activant une transition.

L'objectif principal de la représentation des variables «indicateurs» dans un bloc action est d'augmenter la clarté et la documentation du programme.

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 links have to be programmed explicitly, for example the link through the transition condition T13 in figure 19. 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.

At the conclusion of a scan, the state of the evolution of an SFC action is determined in exactly the same way as any other SFC, that is, by the states of its active and (possibly) contained actions. The execution of an SFC action thus does not involve a complete run through the program until a defined end; rather, it means taking the appropriate actions and preparing for the next scan.

NOTE Care should be taken not to include in an SFC action any edge-activating or -detecting elements that may fail to function properly, if the event can occur while the SFC action is de-activated. Examples of such elements include delayed (D) actions, timers, etc.

Although an SFC action is not a program organization unit, recursive SFC actions should not be used, since this could in principle cause an infinite nesting of SFC scans.

3.9.5 SFC function blocks

Not only an action but also the body of a function block can be defined by an SFC; this will be denoted as an *SFC function block*. The use of such function blocks follows the rules common to all function blocks. The user can declare several instances of the same function block type; each will have a private data area, which also contains information about the current state of the SFC. Each instance then can work independently, using the same SFC algorithm but with separate, hidden internal state variables.

A declaration of a simple SFC function block is shown in figure 20. It will be noted that the SFC contained in an SFC function block is defined in the same (textual or graphical) way and under the same restrictions as for ordinary SFCs. In particular, each SFC network of an SFC function block shall contain an initial step.

Instances of SFC function blocks can be invoked in the same manner as all other function blocks. Input variables that control the evolution of the SFC can be declared and used. For example, a Boolean transition condition for starting the evolution can be passed via input variables such as T11 in figure 20. 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.

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.

L'utilisateur programme une variable «Indicateur» dans le contenu d'une action, par exemple comme le montre la figure 19.

Comme on programme librement les variables «indicateur», on peut les utiliser de manière très souple. On peut, par exemple, associer la variable à la première occurrence d'un état défini. La variable peut aussi être associée à la satisfaction d'une condition pendant une longue période de temps ou un état disponible différé. Cette flexibilité serait perdue si la fonctionnalité de la variable «indicateur» était définie en tant que partie de la bibliothèque d'un automate programmable.

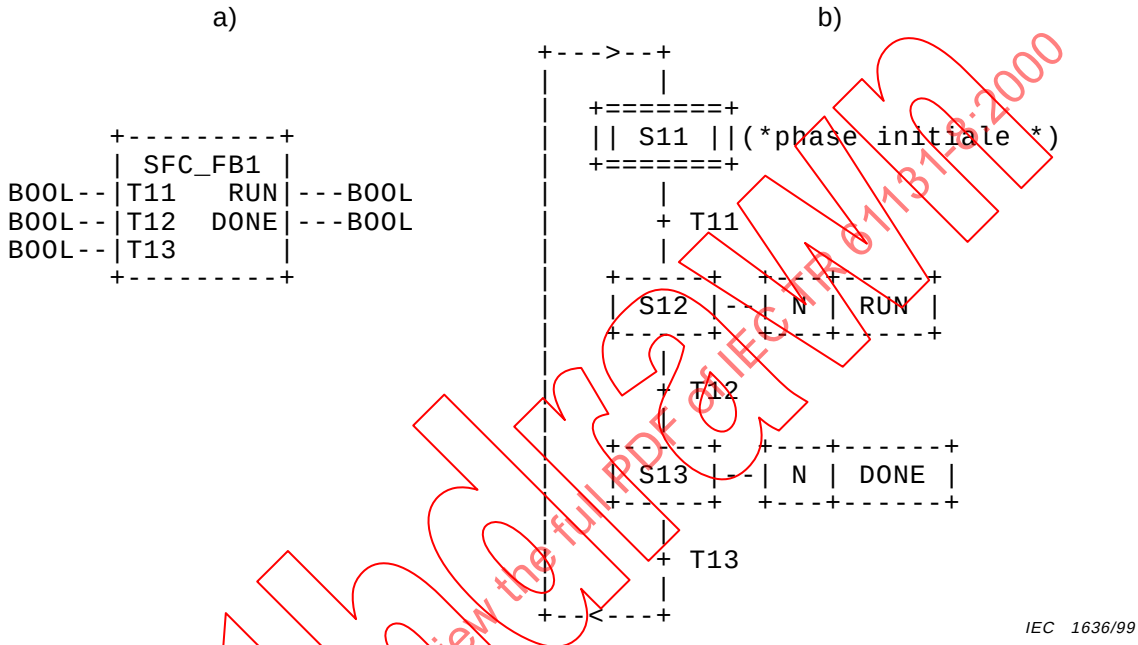


Figure 20 – Bloc fonctionnel SFC
a) Interface externe
b) Corps

3.10 Mécanismes d'ordonnancement, de concurrence et de synchronisation

3.10.1 Problèmes des systèmes d'exploitation

Les automates programmables ont toujours servi à commander en temps réel une multitude de processus industriels parallèles. Dans le passé, les stratégies d'exécution des programmes supposaient que l'automate pouvait effectuer son programme en entier aussi vite que cela était nécessaire pour respecter les contraintes de temps réel du processus à commander le plus rapide. Cette approche est facile à mettre en oeuvre et à utiliser, et elle est appropriée à beaucoup de systèmes. Un seul automate peut, cependant, n'être plus capable de satisfaire à des exigences toujours plus grandes en matière de vitesse et de complexité avec un balayage direct d'un seul programme.

La CEI 61131-3 offre des éléments de langage supplémentaires pour aider l'utilisateur à commander l'exécution des programmes pour satisfaire à ces nouvelles exigences. Afin d'aider les programmeurs à utiliser de manière plus efficace ces nouveaux dispositifs, les termes les plus importants sont expliqués ci-dessous.

The programming of an “indicator” variable is done in the content of an action by the user; an example of this is shown in figure 19.

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.

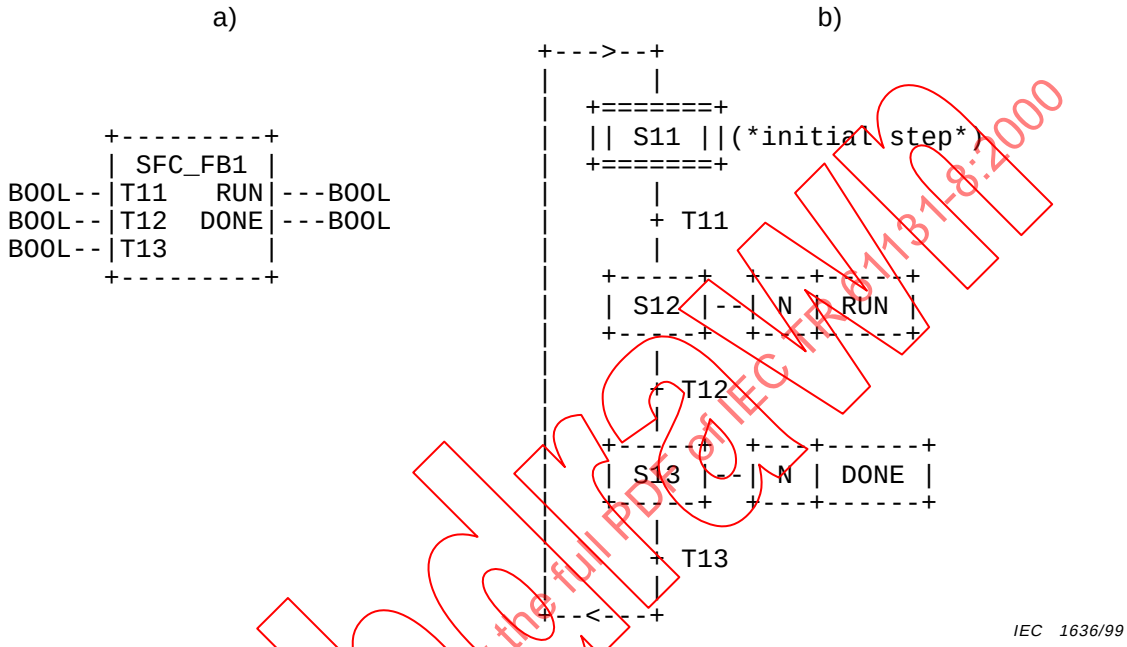


Figure 20 – An SFC-function block
a) External interface
b) Body

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 could 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 use 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.

Dans cette explication, le terme TASK (tâche) se réfère à un ensemble de programmes ou de blocs fonctionnels qui s'exécute de façon indépendante des autres tâches et de façon (quasi) parallèle à ces dernières.

Le terme TASK sert de mot clé dans la CEI 61131-3 pour définir l'activation temporelle d'une tâche, c'est-à-dire, l'instruction TASK spécifie les dispositions d'exécution des tâches associées. Le court exemple ci-dessous est tiré de la figure 20 de la CEI 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);
```

Les caractéristiques des quatre tâches sont explicitement définies dans les quatre premières lignes. Les trois premières lignes définissent les caractéristiques des tâches dont l'exécution doit être planifiée périodiquement, aux intervalles spécifiés, alors que la quatrième ligne spécifie une tâche déclenchée par le front montant d'une variable booléenne «z2».

Le programme P1 et le bloc fonctionnel FB1 dans le programme P2 sont donc associés avec une tâche définie comme étant SLOW_1, qui s'exécute toutes les 20 ms avec une priorité 2. La tâche plus rapide FAST_1 ne comporte que le bloc fonctionnel FB2. Le programme P2 appartient à une tâche qui est toujours active quand il n'y a pas d'autres tâches exécutables (voir 2.7.2 de la CEI 61131-3). Le programme P4 est associé à la tâche INT-2, déclenchée par le front montant de la variable «z2», alors que le bloc fonctionnel FB1 en P4 est associé à la tâche périodique PER_2.

La stratégie de sélection des tâches pour l'exécution détermine les temps de réponse du système et l'efficacité d'utilisation de ses capacités de traitement. Temps de réponse et efficacité sont en général améliorés quand une tâche en cours d'exécution peut être interrompue par une autre («ordonnancement préemptif»). Ces aspects sont traités en 3.10.2.

D'autres issues relatives au multitraitement sont le contrôle d'un accès exclusif à l'équipement d'exploitation et les dispositions nécessaires au transfert de données entre les tâches. Ces aspects sont traités respectivement aux paragraphes 3.10.3 et 3.10.4.

Comme on le voit dans le tableau 2, les systèmes multi-utilisateurs, multi-tâches tels que UNIX et les temps de réponse et l'efficacité en temps réel sont, en général, mutuellement exclusifs.

In this explanation, 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 the four tasks are defined explicitly by the first four lines. The first three lines define the characteristics of tasks whose execution is to be scheduled periodically at the specified intervals, whilst 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”, whilst 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 subclauses 3.10.3 and 3.10.4, respectively.

As shown in table 2, multi-user/multitasking systems such as UNIX and real-time responsiveness and efficiency are usually mutually exclusive.

Tableau 2 – Différences entre systèmes multi-utilisateurs et systèmes en temps réel

Systèmes multi-utilisateurs	Systèmes en temps réel
Une bonne distribution du temps du processeur entre les tâches en cours d'exécution génère un bon taux de productivité du système.	La tâche en cours d'exécution doit tenir compte des processus externes. Une réaction rapide aux événements extérieurs et l'exécution de la tâche réponse avant une nouvelle occurrence sont nécessaires.
Les temps d'exécution des programmes ne peuvent pas être calculés.	Les limites des temps d'exécution des programmes (par exemple les temps de réponse) doivent être garanties.
La gestion de la mémoire peut envoyer un programme sur une mémoire secondaire (allocation dynamique de ressources).	L'allocation des ressources est fixe. (Le rechargement des tâches stockées sur une mémoire secondaire prend du temps.)
La priorité des tâches est modifiée dynamiquement, par exemple en fonction de l'utilisation du temps unité centrale.	Les priorités sont fixes. La tâche avec la plus forte priorité obtient le processeur aussi longtemps qu'elle le désire. La tâche ne peut être interrompue que par une tâche de priorité supérieure ou quand elle rend explicitement le contrôle.
Les fichiers de données sont généralement dans des bibliothèques à structures en arbre. De gros fichiers peuvent être distribués sur des secteurs non contigus du disque.	L'exigence pour des entrées/sorties disque rapides impose le stockage des données sur des secteurs contigus.
Le débit du bus est susceptible d'être un goulot d'étranglement.	L'interruption de réponse du bus est susceptible d'être un goulot d'étranglement.

3.10.2 Ordonnancement des tâches

Le paragraphe 2.7.2 de la CEI 61131-3 introduit deux méthodes d'ordonnancement de l'exécution des tâches, dénommées ordonnancement préemptif et non préemptif. Les deux méthodes sont employées dans de nombreux systèmes en temps réel. L'ordonnancement préemptif est le plus souvent utilisé dans les grands automates programmables et les ordinateurs de processus, alors que l'on trouve l'ordonnancement non préemptif dans les automates programmables plus petits.

Un ordonnancement non préemptif suppose que le processeur d'un automate programmable ne peut passer d'une tâche à une autre (généralement appelé commutation de contextes) qu'après l'exécution complète de toutes les POU associées à la tâche en cours. Après l'exécution complète de ces POU, la prochaine tâche à être activée sera celle qui a la plus forte priorité ou celle qui a attendu le plus longtemps quand plusieurs tâches ont le même niveau de priorité. L'ordonnancement préemptif permet une commutation de contexte à n'importe quel moment. Dès qu'une tâche d'une priorité supérieure à celle de la tâche en cours est activée, en fonction des entrées de bloc TASK associé, l'état complet de la tâche en cours d'exécution sera sauvé par le processeur et la tâche sera suspendue. On exécutera ensuite la tâche de priorité supérieure. On appelle cette procédure «commutation préemptive de contexte», car le processeur (ou au moins tous ses registres) doit être vidé en faveur de la tâche de priorité supérieure. La nouvelle tâche est supposée devoir libérer le processeur plus tard. Après cela, la tâche suspendue sera replacée dans son contexte (toutes les valeurs de registres restaurées à leurs valeurs d'avant la préemption) et reprendra son exécution. L'ordonnancement préemptif peut avoir des contextes imbriqués: il peut y avoir, à un moment donné, plusieurs tâches de priorité plus basse qui ont été suspendues.

3.10.2.1 Effets sur les performances

Les deux méthodes d'ordonnancement (préemptive et non préemptive) essaient de faire exécuter les tâches en attente de plus haute priorité avant les tâches dont la priorité est de niveau plus bas. La principale différence est le temps d'attente d'une tâche de priorité supérieure quand elle a demandé son exécution. Avec ordonnancement préemptif, ce temps est habituellement très court. En fonction du système d'exploitation sous-jacent et de la vitesse du processeur, le temps nécessaire à la prise en charge d'une commutation de contexte se

Table 2 – 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 are necessary.
Program running times cannot be calculated.	Based on program running times (e.g. 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, e.g. 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 disk sectors.	High disk 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, whilst 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 program organization units associated with the current tasks. After complete execution of all such program organization units, 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 (i.e. the time from setting a task ready to execute by

situe dans une plage entre quelques microsecondes et des millisecondes. Cela implique que le temps de réaction du système (c'est-à-dire le temps entre le moment où la tâche est mise à l'état prêt à l'aide de ses blocs TASK et le commencement de l'exécution) qui utilise l'ordonnancement préemptif est court, malgré le coût de quelques traitements supplémentaires pour sauver et restaurer le contexte de données.

Quand un automate programmable non préemptif passe d'une tâche à une autre, il n'y a, dans le processus, qu'une quantité de données négligeable à sauver. La durée de la commutation de contexte elle-même est courte, probablement plus courte que dans le cas d'un contexte d'ordonnancement préemptif; mais le passage d'une tâche à une autre ne s'effectue jamais pendant qu'une POU est en cours d'exécution. En conséquence, avec un ordonnancement non préemptif, le temps de réaction le plus défavorable est au moins égal à la durée du programme ou du bloc fonctionnel le plus long à arriver à la fin de l'exécution. Etant donné que cela ne s'applique pas seulement aux blocs fonctionnels du système, mais aussi à tous les programmes et blocs fonctionnels définis par l'utilisateur, il peut être difficile de déterminer les limites du temps de réaction d'un système à ordonnancement non préemptif.

NOTE On peut dans certaines applications, avoir besoin d'utiliser d'autres systèmes à ordonnancement que ceux définis dans la CEI 61131-3, qui garantissent des temps maximaux de réaction.

La commutation préemptive des tâches cause habituellement une légère dégradation des performances (car la commutation de contexte prend du temps) mais elle améliore les temps de réaction de façon significative. On peut voir cela en comparant la colonne 2 de l'exemple 1 du tableau 50 en 2.7.2 de la CEI 61131-3, avec la colonne correspondante de l'exemple 2. On peut y voir que dans le cas de l'ordonnancement préemptif (exemple 2) le bloc fonctionnel de forte priorité FB2 dans le programme P2 (P2.FB2@1) n'a jamais à attendre, alors que, dans le cas d'un ordonnancement non préemptif (exemple 1), il doit attendre 4 s en deux ou trois activations (à t = 10 ms et t = 20 ms).

3.10.2.2 Effets de la concurrence

Satisfaire les règles de concurrence fournies au point 7) de 2.7.2 de la CEI 61131-3, peut être sensiblement plus difficile dans un système d'automates programmables avec un ordonnancement préemptif de tâches que dans un système à ordonnancement non préemptif. Dans certaines configurations et pour certains programmes, un ordonnancement préemptif peut ne pas être capable de garantir que ces règles sont respectées. Le fabricant de l'automate programmable fournira suffisamment d'informations pour permettre à l'utilisateur de déterminer par quels moyens et dans quelle mesure ces règles sont satisfaites.

3.10.3 Sémaphores

3.10.3.1 Généralités

Les sémaphores servent à des fins de contrôle d'accès à certains équipements utilisés communément (disques, imprimantes, etc.). Une seule tâche à la fois peut accéder à ces ressources; les autres accès sont rejetés ou différés conformément au principe que le programme commence par tester le sémaphore pour savoir si la ressource n'est pas déjà affectée; si elle ne l'est pas, le programme la demande. L'affectation réussie d'un sémaphore à une tâche donne à cette tâche le droit d'accéder aux données ou à l'équipement associés. Le sémaphore est libéré après la fin de l'exécution.

Conformément à 3.3.2.4 de la CEI 61131-3, il ne faut pas utiliser les instructions «WHILE et REPEAT» pour synchroniser des processus, par exemple pour une boucle d'attente avec une condition de fin externe. Il faut utiliser pour cela les éléments SFC définis en 2.6. Cette interdiction s'applique aux sémaphores qui sont aussi des mécanismes de synchronisation de processus. Transgresser cette règle mène à une grave dégradation de la fiabilité, de la portabilité et de la réutilisabilité du logiciel; dans le cas de sémaphores, cela peut même conduire à des suspensions de programmes prématurées et indécelables pendant l'exécution d'une POU (unité d'organisation de programmes).

means of its TASK block to the beginning of execution) of a system using preemptive scheduling is short, at the expense of some additional processing time to save and restore context data.

When a non-preemptive programmable controller switches from one task to another, there is a negligible amount of data in the processor that has to be saved. The context switch itself would be short, probably shorter than in the case of a preemptive context switch; but this changeover from one task to another will never be done while a program execution unit 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 function blocks but also for all user-defined function blocks 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 other scheduling mechanisms 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 2.7.2 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, whilst with non-preemptive scheduling (example 1) it has to wait for 4 s in two out of three activators (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

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 sever 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.

La figure 13 en 2.5.3.1 de la CEI 61131-3 donne un exemple d'utilisation d'éléments SFC en liaison avec un bloc fonctionnel sémaphore (SEMA). C'est un cas limité d'utilisation du concept informatique de sémaphore qui prescrit la suspension d'une tâche qui attend un sémaphore et la reprise de l'exécution après la libération du sémaphore. La CEI 61131-3 impose volontairement cette limitation afin de fournir un niveau plus élevé de maintenabilité: le fait qu'une tâche attende un sémaphore est directement visible sur l'affichage SFC pendant le fonctionnement.

Le sémaphore informatique met en oeuvre typiquement les instructions «WAIT» et «SIGNAL» où le sémaphore permet un certain nombre d'accès simultanés. Un contrôleur de clavier permet, par exemple, de prendre jusqu'à 20 caractères. Une tâche peut donc utiliser «SIGNAL» 20 fois avant que l'accès soit rejeté. Pour la maintenabilité, cela est réduit à un «CLAIM» et un «RELEASE» uniques dans le tableau 34 en 2.5.2.3.1 de la CEI 61131-3.

3.10.3.2 Blocages

Un usage incorrect des sémaphores peut aboutir à ce qu'une tâche attende un équipement opérationnel utilisé par une autre tâche et vice versa. La conséquence est que les deux tâches vont en principe attendre indéfiniment, c'est ce qu'on appelle un blocage (deadlock).

Il y a quatre conditions pour un blocage:

1) les tâches bloquées ont un accès exclusif à l'équipement opérationnel correspondant;
2) les tâches bloquées ont déjà reçu l'affectation de l'équipement opérationnel et elles attendent l'affectation d'autres équipements;
3) il n'est pas possible d'annuler l'affectation d'un équipement à une tâche avant qu'elle ne la libère elle-même;
4) il y a une chaîne fermée de tâches, dans laquelle chaque tâche a reçu l'affectation d'un équipement attendu par la tâche voisine dans la chaîne.

Il convient que les PSE (environnement de support de programmation) pour les automates programmables parviennent à fournir des méthodes pour éviter, prévenir, détecter et/ou supprimer les blocages. En attendant, il convient que les programmeurs et les concepteurs de systèmes soient avertis de l'existence potentielle de blocages et qu'ils utilisent le bloc fonctionnel SEMA pour le contrôle des accès partagés à des équipements opérationnels. Cela simplifiera la recherche des blocages quand ils surviennent et l'application future des algorithmes de prévention des blocages.

3.10.4 Messagerie

La messagerie (communication entre processus) décrit des méthodes permettant aux tâches d'échanger des données entre elles et (éventuellement) de synchroniser leurs exécutions. Il y a en général trois méthodes pour ces transferts de données, à savoir le stockage global, les boîtes aux lettres et les queues. Il est aussi possible d'utiliser des sémaphores servant à la communication, mais cela peut être considéré comme un cas particulier de boîtes aux lettres.

3.10.4.1 Stockage global

Deux tâches peuvent échanger des données via une zone de stockage accessible à chacune d'elles. Le protocole d'accès doit être explicitement inclus dans le programme. Sauf cette obligation, il reste une grande liberté pour la mise en oeuvre du transfert de données. On met en oeuvre cette facilité par les structures VAR_GLOBAL et VAR_EXTERNAL définies en 2.4.3 et 2.7.1 de la CEI 61131-3.

An example of the use of SFC elements in conjunction with the semaphore (SEMA) function block is given in figure 13 of 2.5.3.1 of IEC 61131-3. This is a limited case of the computer science concept of “semaphore”, which prescribes the suspension of a task waiting for a semaphore and resumption of execution after the semaphore is released. This limitation is intentionally imposed in IEC 61131-3 in order to provide a higher level of maintainability: the fact that the task is waiting for a semaphore is directly visible from the SFC display at run time.

The computer science “semaphore” also typically implements the commands “WAIT” and “SIGNAL” where the semaphore allows a certain amount of simultaneous accesses. A keyboard driver, for example, can take up to 20 characters. Thus a task can use “SIGNAL” 20 times before an access will be refused. For maintainability, this is restricted to a single CLAIM and RELEASE in table 34 of 2.5.2.3.1 of IEC 61131-3.

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:

1) the deadlocked tasks have exclusive access to the corresponding operational equipment;
2) the deadlocked tasks already have operational equipment assigned to them while they wait for further operational equipment;
3) it is not possible to remove the assignment of operational equipment to any task until the task releases it;
4) 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 Programming Support Environments (PSEs) should 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 deadlock and should use the SEMA function block to control shared access to operational equipment. This will 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 Boîtes aux lettres et queues

Une boîte aux lettres est une sorte de boîte pour les données à transférer. Contrairement au transfert via un stockage global, il existe un protocole de transfert des données vers la boîte aux lettres et pour les rendre ensuite disponibles aux autres partenaires. Le transfert des données peut s'effectuer au moyen de divers supports tels que des zones communes de stockages, fonds de paniers ou réseaux de communication.

La caractéristique des queues est que les éléments de données sont lus dans l'ordre où ils y sont entrés. Le tampon d'un terminal en est un exemple, il contient les caractères dans l'ordre de leur arrivée.

Les boites aux lettres ou les queues, ou les deux dispositifs, peuvent être utilisés par le bloc fonctionnel de communication décrit en 3.11. Le mécanisme particulier employé est cependant invisible à l'utilisateur, qui n'est concerné que par les interfaces externes et les fonctionnalités de ces blocs fonctionnels.

3.10.5 Tampons horaires

Le terme «tampon horaire» vient du domaine des bases de données. Grâce aux tampons horaires, chaque élément de données de la base de données ne contient pas seulement la valeur de la donnée, mais aussi la date de son introduction. Si la donnée change, l'ancien élément n'est pas retiré de la base de données, mais il est signalé comme n'étant plus valable et le nouvel élément est introduit avec son tampon horaire. L'avantage de cette approche est que l'on peut détecter assez facilement les changements dans la base de données en suivant les tampons horaires; on peut aussi supprimer, dans la base de données de nouvelles valeurs quand elles sont erronées et restaurer les anciennes qui sont correctes.

Les tampons horaires sont aussi intensivement utilisés par les systèmes de commande pour les processus industriels continus pour des raisons similaires, à savoir, pour mettre en place un historique de l'enregistrement des données du processus, pour déterminer la validité des données en fonction de leur âge et pour restaurer les données erronées à une valeur précédemment correcte.

Le tamponnage horaire des données est facilement supporté en incluant la valeur de la date des données dans une variable de type donnée structurée qui inclut également son propre tampon horaire. Un type de donnée avec tampon horaire contenant une valeur REAL peut, par exemple, être déclaré en suivant les règles définies en 2.3.3 de la CEI 61131-3 comme:

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


3.11 Facilités de communications dans l'ISO/CEI 9506-5 et la CEI 61131-5

La CEI 61131-5 définit un jeu de blocs fonctionnels normalisés qui pourra être utilisé par les langages de la CEI 61131-3 pour la communication entre les automates programmables, ainsi que pour la communication avec les dispositifs de l'hôte. La CEI 61131-5, en plus, décrit la correspondance de l'exploitation de ces blocs fonctionnels avec les services de MMS comme exemple de techniques générales de leur application sur tout ensemble de services sous-jacents. Cela permet à l'utilisateur de programmer des fonctionnalités de communications indépendantes des réseaux dans tout système d'automates programmables conforme à la CEI 61131-3 et à la CEI 61131-5.

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 done via various media, e.g. 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. 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.

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 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

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. In addition, IEC 61131-5 describes the mapping of the operation of these function blocks onto MMS services as an example of the general technique for their mapping onto any set of underlying services. This enables the user to program network-independent communication functionality in any programmable controller system complying to IEC 61131-3 and IEC 61131-5.

L'ISO/CEI 9506-5 spécifie les exigences pour un automate programmable servant de VMD (virtual manufacturing device = dispositif virtuel de fabrication) dans le contexte MMS (spécification de messagerie industrielle), y compris l'application de certains éléments de la CEI 61131-3 dans le contexte MMS.

3.11.1 Canaux de communication

Un automate programmable peut établir une communication avec un dispositif éloigné d'un automate programmable en se servant de la CEI 61131-5 pour ouvrir un *canal* de communications.

Les logiciels des couches basses de la pile de protocoles de réseau garantissent que les données sont transférées vers le noeud éloigné et à partir de ce noeud, si possible sans erreurs. Par exemple, si une erreur survient pendant la communication et que les données sont détériorées, elles peuvent être automatiquement retransmises.

Tous les détails relatifs au commande de bas niveaux des réseaux tels que queue de messages, préparation des trames, etc. sont cachés par les blocs fonctionnels de la CEI 61131-5. Les programmeurs d'applications ne sont concernés que par les aspects spécifiques aux applications des canaux.

Les spécifications de l'ISO/CEI 9506-5 et de la CEI 61131-5 ne permettent qu'un seul niveau hiérarchique pour l'adressage des objets chargeables, tels que les programmes de la CEI 61131-3 dans un automate programmable éloigné. En conséquence, il suffit d'un unique nom d'instance de programme au sein d'une configuration pour garantir un adressage unique de programme à distance via un canal donné, bien que le domaine des ressources permette d'utiliser des noms identiques dans différentes ressources.

3.11.2 Lire et écrire des variables

Un automate programmable peut lire et écrire dans des variables dans d'autres automates programmables éloignés conformes à la CEI 61131-5, à condition qu'elles soient des variables représentées directement telles que définies en 2.4.1.1 de la CEI 61131-3, ou des variables déclarées pour être accessibles en utilisant la construction VAR_ACCES... END_VAR définie en 2.7.1 de la CEI 61131-3. Ces lectures ou écritures sont respectivement réalisées à l'aide des blocs fonctionnels READ ou WRITE, comme décrit ci-dessus.

Il est recommandé que la méthode VAR_ACCESS soit toujours utilisée quand on accède à des variables dans des automates programmables éloignés, car il est alors possible d'utiliser des noms significatifs pour les variables. Il y a toujours un risque que l'adresse physique I/O soit modifiée si le programme de l'automate programmable éloigné est modifié. Il peut être pratique de fixer les noms de VAR_ACCESS pour plusieurs programmes différents d'automates programmables.

3.11.3 Blocs fonctionnels de communication

La CEI 61131-5 définit un jeu de blocs fonctionnels que l'on peut commander et auxquels on peut accéder à partir d'un programme de la CEI 61131-3 de la même manière que pour tous les autres blocs. Ces blocs fonctionnels fournissent les fonctionnalités suivantes:

CONNECT (connecter): fournit un «channel ID» (identification de canal) local (pas un identificateur au sens de la CEI 61131-3) pour communiquer avec un dispositif éloigné. Il convient que le dispositif éloigné ait un nom unique. Le channel ID fourni par ce bloc fonctionnel peut être utilisé par d'autres blocs fonctionnels de communication pour identifier des dispositifs éloignés.

STATUS (état): interroge un dispositif éloigné pour une vérification des informations concernant ce dispositif. Il est important pour un automate programmable de vérifier périodiquement l'état d'un dispositif éloigné afin de s'assurer que cet automate programmable se comporte correctement.

ISO/IEC 9506-5 specifies the requirements for a programmable controller serving as a virtual manufacturing device (VMD) in the manufacturing message specification (MMS) context, including the mapping of certain IEC 61131-3 elements into the MMS context.

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 function blocks, 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.

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 information for device verification. 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 (état utilisateur): permet à un automate programmable éloigné de recevoir les informations de vérification d'un dispositif éloigné, y compris celles concernant son état physique et logique. Le dispositif éloigné doit avoir la capacité d'envoyer les informations de vérification, chaque fois qu'elles changent.

READ (lire): interroge un dispositif éloigné pour obtenir les valeurs d'une ou plusieurs variables. On peut donner une liste de variables comme entrées pour ce bloc. Après un délai assez court dû au temps de transfert de la demande et de la réponse à travers le réseau, les valeurs des variables éloignées sont présentées en tant que sorties de bloc fonctionnel.

NOTE Le bloc READ ne fournit pas de paramètre d'entrée pour contrôler la vitesse d'interrogation. Le programme d'application redéclenchera le bloc pour initialiser une nouvelle interrogation.

WRITE (écrire): écrit une ou plusieurs valeurs dans une ou plusieurs variables dans un automate éloigné. On peut spécifier une liste de noms de variables pour identifier les variables dans le dispositif éloigné. Le dispositif éloigné est sélectionné par le paramètre R_ID obtenu par le bloc CONNECT.

USEND (utilisateur final): envoie les valeurs d'une ou plusieurs variables à un bloc URCV dans un programme d'application à distance. Le programme d'application peut utiliser les valeurs transférées dans le bloc URCV d'une façon normale. Un paramètre R_ID garantit que le bloc USEND local envoie les valeurs au bon bloc URCV dans le dispositif éloigné.

URCV (réception utilisateur): reçoit les valeurs d'une ou plusieurs variables à partir d'un bloc USEND associé.

SEND (envoyer): fournit un échange entrecroisé de données avec un bloc RCV dans un dispositif éloigné. Le bloc SEND envoie les valeurs d'une ou plusieurs variables à un block RCV éloigné correspondant à un «channel ID» d'un bloc CONNECT et à un paramètre R_ID. Quand il reçoit ces valeurs, le programme d'application de l'automate programmable éloigné charge comme réponse un ensemble de valeurs qui est renvoyé au bloc SEND. Les blocs SEND et RCV sont fournis pour des applications demandant des échange entrecroisés ou des échanges de données entre programmes d'application locaux et les programmes éloignés.

RCV (recevoir): reçoit les valeurs d'une ou plusieurs variables à partir d'un bloc SEND associé.

NOTE Le temps entre l'indication NDR (new data ready = nouvelles données prêtes) du bloc RCV et la demande de transmission de la réponse est déterminé par le programme d'application. Comme, dans beaucoup de systèmes de communication, le canal peut se bloquer s'il y a trop de réponses en attente, on recommande des structures de programmes qui répondent aussi vite que possible au signal NDR.

ALARM (alerte): envoie une valeur d'une ou plusieurs variables à un dispositif éloigné identifié par le channel ID avec identificateur d'un changement d'état quand une condition de changement d'état a été détectée. Cette alerte peut être caractérisée par un niveau de sévérité. Ce bloc s'attend à ce que le dispositif éloigné émette un accusé de réception de l'alerte.

NOTIFY (notification): semblable au bloc ALARM mais on n'attend pas d'accusé de réception de la part du dispositif éloigné.

3.12 Pratiques de programmation recommandées

Il convient que les effets des techniques de programmation sur la qualité du logiciel soient considérés quand on choisit parmi les options disponibles dans la CEI 61131-3. Le présent paragraphe signale quelques uns des effets les plus significatifs et recommande des pratiques de programmation pour obtenir une meilleure qualité des logiciels.

3.12.1 Variables globales

Un usage excessif des variables globales contredit les principes d'encapsulage et de masquage exposés en 2.4.2.1, et peut grandement réduire la fiabilité, la maintenabilité et la réutilisabilité du logiciel. Il faut, en particulier, éviter d'écrire des variables globales de plus d'un endroit du programme. Il est recommandé que les variables globales soient utilisées tout au plus pour:

USTATUS: allows a remote 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 parameter 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 programmable controller. A list of variable names can be specified to identify variables in a remote device. The remote device is selected by the R_ID parameter 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. A R_ID parameter 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 a 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 parameter. 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: send 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 a 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.

3.12 Recommended 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:

– définir des chemins d'accès pour une communication ouverte;

– fournir des valeurs d'intérêt global pour d'autres POU.

Il convient que les variables globales ne servent jamais à communiquer des données entre des programmes exécutés de façon asynchrone si la concurrence pour les données est un problème important. Les blocs fonctionnels SEND/RCV ou USEND/URCV seront utilisés pour traiter la concurrence.

3.12.2 Sauts dans le langage FBD (blocs fonctionnels)

Comme on le note en 2.3, le langage FBD est modélisé sur des circuits matériels (IC), c'est-à-dire que les instances dans les réseaux de blocs fonctionnels sont modélisés pour travailler en parallèle. Dans ce contexte, les sauts par dessus ou entre les réseaux peuvent troubler l'utilisateur qui pourrait penser en termes de matériel et, par là, réduisent la fiabilité, la maintenabilité et l'adaptabilité du logiciel. Il est recommandé que l'exécution conditionnelle de réseaux FBD soit contrôlée en plaçant ces réseaux dans des actions SFC comme on le décrit en 2.6.4 de la CEI 61131-3, ou en utilisant l'entrée EN (enable = activer) et la sortie ENO des fonctions décrites en 2.5.1.2 de la CEI 61131-3. Cela correspond mieux au concept d'activation ou désactivation conditionnelle de portions de circuits matériels; l'opération peut alors être plus facilement vérifiée qu'en localisant et en vérifiant l'état des conditions des sauts.

3.12.3 Invocations multiples d'instances de blocs fonctionnels en FBD

Pour les mêmes raisons que celles données ci-dessus, il convient que le corps d'une POU (unité d'organisation de programmes) écrit en langage FBD ne contienne pas de multiples copies de la même instance de bloc fonctionnel. Les utilisateurs orientés vers le matériel ne comprendront pas les diagrammes de circuits contenant plusieurs copies de la même «puce» et il peut aussi être impossible de déterminer la séquence d'exécution de ces multiples copies de la même instance de bloc fonctionnel, ce qui réduit la fiabilité et la maintenabilité du logiciel.

3.12.4 Couplage de réseaux SFC (diagramme fonctionnel en séquence)

Quand il y a de multiples réseaux SFC dans le corps d'un programme ou d'un bloc fonctionnel, ils sont typiquement emboîtés d'une certaine façon. On donne un exemple de cela dans le programme AGV illustré à l'article F.8 de la CEI 61131-3 où l'emboîtement est réalisé à l'aide de la variable booléenne CYCLE et des drapeaux de phase READY.X et DONE.X.

Il peut y avoir des effets inattendus dans les couplages d'états de machines de tous types, y compris des SFC. Un des plus fréquents est le «deadly embrace» (étreinte mortelle), analogue à la condition de blocage (deadlock) décrite en 3.10.3.2, où chaque SFC attend une condition annulant une transition par un autre SFC dans une chaîne bloquée. On recommande les mesures suivantes pour mieux éviter et diagnostiquer de telles conditions, améliorant ainsi la fiabilité et la maintenabilité du logiciel.

1) Eviter l'usage de drapeaux de phase dans les conditions de transition pour détecter la fin de l'exécution d'actions associées à d'autres phases, plus spécialement quand les qualificateurs des actions ne sont pas «N» (non stocké).

2) Quand c'est possible, éviter d'utiliser des actions SFC telles que décrites en 3.9.4, car elles introduisent des couplages via le mécanisme de commande des actions en plus des couplages possibles au travers de conditions de transition.

3) Chaque fois que c'est possible, encapsuler les réseaux individuels SFC dans des blocs fonctionnels SFC, comme on le montre en 3.9.5, et utiliser un bloc fonctionnel diagramme pour exprimer le couplage entre les blocs fonctionnels SFC. Ce couplage explicite renforce considérablement la lisibilité et, par là, la maintenabilité du logiciel, et il fournit aussi le potentiel pour une réutilisation des SFC encapsulés à l'aide du mécanisme d'instanciation des blocs fonctionnels.

– 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 function blocks should be used in such cases to assure concurrency.

3.12.2 Jumps in function block diagram (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 sequential function chart (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, where each SFC is waiting for a transition clearing condition from another SFC in a deadlocked chain. The following measures are recommended to enhance the avoidance and diagnosis of such conditions, thus increasing software reliability and maintainability.

1) 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.

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

3) Whenever possible, encapsulate individual SFC networks into SFC *function blocks* as described in 3.9.5, and use a function block diagram 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.

4) S'assurer que l'on peut atteindre chaque bloc fonctionnel, c'est-à-dire qu'il existe un chemin clos depuis l'étape initiale vers toutes les autres phases dans le réseau avec retour vers l'étape initiale, et que ce chemin est «sûr», c'est-à-dire qu'une génération incontrôlée de jetons n'est pas possible (voir 2.6.5 de la CEI 61131-3 pour plus d'explications sur ce point).

3.12.5 Modification dynamique de la priorité des tâches

Il est bien connu d'après la théorie des systèmes d'exploitation que la modification dynamique de la priorité des tâches peut avoir des effets imprévisibles sur l'exécution des programmes parallèles, y compris la génération de blocages. Comme les TASKS de 2.7 de la CEI 61131-3 ne correspondent pas forcément directement à des tâches du système d'exploitation, il peut se produire des effets supplémentaires dépendant de la mise en oeuvre. Il est donc fortement recommandé que le programmeur d'applications n'utilise pas de modification dynamique de priorités des tâches dans un système conforme à la CEI 61131-3.

3.12.6 Commande de l'exécution des instances de blocs fonctionnels par les tâches

L'association des tâches avec les instances de blocs fonctionnels, et ses effets sur la concurrence pour les données, sont décrits en 2.7.2 de la CEI 61131-3. Il convient que le programmeur soit averti que le fait d'utiliser ces dispositifs peut produire des erreurs dans la consistance des données pendant l'exécution du programme. Il faut consulter les directives fournies par l'implémenteur de la CEI 61131-3 pour déterminer les mécanismes prévus pour garantir la consistance des données. Comme ces mécanismes dépendent de l'implémentation, les programmes utilisant ces dispositifs peuvent ne pas être portables sur d'autres systèmes conformes à la CEI 61131-3.

3.12.7 Utilisation des blocs fonctionnels RTC (real time clock = horloge temps réel)

L'accès à la date et à l'heure réelles est fourni par le bloc fonctionnel RTC (horloge temps réel) décrit dans le tableau 37 de la CEI 61131-3. Pour se servir du bloc fonctionnel RTC, il faut au préalable créer une instance du type bloc fonctionnel RTC dans une POU, par exemple dans le corps d'un programme ou d'un bloc fonctionnel.

On peut mettre la RTC à l'heure en invoquant l'instance de bloc fonctionnel avec l'entrée EN à 1 (TRUE), à condition que le bloc fonctionnel ait déjà été invoqué avec EN à 0 (FALSE) ou que le bloc fonctionnel soit invoqué pour la première fois. Le paramètre PDT (preset date and time = date et heure prédéterminées) recevra la date et l'heure initiales. A noter que, pour remettre à nouveau la RTC à l'heure, il faut invoquer l'instance RTC avec EN à 0 puis avec EN à 1.

On peut lire l'heure réelle en invoquant le bloc fonctionnel RTC avec l'entrée EN à 1. Le paramètre de sortie CDT (current date and time = date et heure réelles) fournira la date et l'heure au moment où le bloc fonctionnel a été invoqué. La valeur du paramètre de sortie CDT restera inchangée pour la dernière invocation tant qu'on n'aura pas appelé à nouveau le bloc fonctionnel RTC, c'est-à-dire que la valeur ne changera pas entre deux invocations du bloc fonctionnel. Le comportement du paramètre de sortie CDT de la RTC, quand l'entrée EN est à 0, n'est pas spécifié.

Certaines mises en oeuvre d'automates programmables ne peuvent supporter qu'une seule instance de la RTC, ce qui, déclaré comme variable globale, est suffisant pour que l'on dispose de la date et de l'heure partout dans le programme. D'autres mises en oeuvre peuvent autoriser la création de multiples instances de la RTC, où chaque instance de la RTC se comportera comme une horloge autonome. Chaque instance fournira une sortie CDT telle qu'elle est déterminée par son propre paramètre d'entrée PDT. Cela peut être utile quand on traite des problèmes d'heures entre différents fuseaux horaires. Une RTC, par exemple, peut être préréglée à l'heure locale, et une autre RTC à celle d'un autre fuseau horaire.

4) 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 this path 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 in 2.7 of IEC 61131-3 TASKS are not necessarily mapped directly to operating system tasks, additional implementation-dependent effects may occur. Therefore, it is highly recommended to the application programmer not to 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 Use of RTC (real time clock) function blocks

Access to the current data and time is provided by the real time clock (RTC) function block, as described in table 37 of IEC 61131-3. To use the RTC function block it is first necessary to create an instance from the RTC function block type within a POU, for example within a program or function block body.

The RTC can be set to the current time by invoking the function block instance with the enable input EN set to 1 (TRUE), provided that either the function block has been previously invoked with the enable EN set to 0 (FALSE), or the function block is being invoked for the first time. The parameter PDT (preset date and time) should be set to the initial date and time. Note that, in order to reset the RTC again, it is necessary to invoke the RTC instance with EN set to 0 and then invoke with EN set to 1.

The current time can be read by invoking the RTC function block with the input enable EN set to 1. The output parameter CDT (current date and time) will provide the date and time valid at the time the function block is invoked. The value of the output parameter CDT will remain unchanged at the value for the last invocation until the RTC function block is called again, i.e. the value does not change between function block invocations. The behaviour of the RTC output parameter CDT while the enable input EN is 0 (FALSE) is not specified.

Some programmable controller implementations may support only one instance of the RTC, which, if instanced as a global variable, is sufficient to allow the current date and time to be available anywhere in the program. Other implementations may permit the creation of multiple RTC instances, where each RTC instance will behave as a separate clock. Each instance will provide a CDT output as determined by its own PDT input parameter. This might be useful when dealing with the time of day across different time zones. For example, one RTC could be set to the local date and time whilst a second RTC instance could be preset with the date and time in a different time zone.

4 Lignes directrices pour la mise en oeuvre

Le présent article décrit les techniques de mise en oeuvre compatibles avec les objectifs de la CEI 61131-3. D'autres mises en oeuvre sont possibles et autorisées, pour autant qu'elles satisfassent les exigences fonctionnelles imposées par la CEI 61131-3.

4.1 Allocation des ressources

La ressource décrite en 1.4 et 2.7.1 de la CEI 61131-3, est l'unité de base de la CEI 61131-3 pour le stockage des données et des codes de programmes, l'ordonnancement de code pour exécution et l'exécution du code approprié quand on invoque une POU (unité d'organisation de programmes), c'est-à-dire un programme, une *fonction*, ou un bloc *fonctionnel*.

Il est possible, lors d'une mise en oeuvre, de stocker dans une ressource une seule copie du code exécutable associé à chaque type de POU résidant réellement dans la ressource. Il faut, d'autre part, fournir une zone de stockage séparée pour chaque instance de bloc fonctionnel ou de programme. Cela inclut des données suffisantes pour toutes les variables et (éventuellement) pour les informations d'état SFC associées au programme ou au bloc fonctionnel. Le stockage des variables pour une fonction est, au contraire, temporaire et n'existe que pendant l'exécution de la fonction; le stockage de ces données est donc typiquement alloué de façon dynamique dans une pile, (premier entré, dernier sorti) ou dans une zone de mémoire réservée pour des allocations temporaires («heap»).

4.2 Mise en oeuvre des types de données

4.2.1 Types de données REAL et LREAL

Afin de réduire la perte de précision survenant dans les calculs en virgule flottante, il peut être nécessaire de convertir toutes les valeurs en virgule flottante stockées avec le type REAL dans le format nombre en double précision (LREAL), avant de lancer une séquence d'opérations arithmétiques, en particulier si ces calculs peuvent traiter de différences relativement petites entre les nombres à virgule flottante. Les résultats sont ensuite reconvertis dans le format REAL pour le stockage.

Il existe une grande variété de microprocesseurs, en particulier les microprocesseurs à signaux numériques DSP (digital signal processor), qui ont leurs propres formats internes de virgule flottante. Dans cette situation, l'implémenteur doit limiter la plage de valeurs supportée par les types REAL et/ou LREAL lors de la mise en oeuvre aux valeurs spécifiées en 2.3.1 de la CEI 61131-3, ou il doit spécifier de nouveaux types de données dépendant de la mise en oeuvre, disons les types DSP_REAL et DSP_LREAL pour les formats des nombres à virgule flottante de la machine, tout en supportant les types normalisés REAL et LREAL, pour rester conforme aux exigences de 1.5.1 a) et h) de la CEI 61131-3.

4.2.2 Chaînes de caractères

La longueur maximale et le format des variables chaînes de caractères dépendent de la mise en oeuvre. Cela implique que les blocs fonctionnels et les algorithmes utilisant des données chaînes de caractères peuvent ne pas être portables entre automates programmables, en particulier si la longueur maximale de la chaîne est significativement différente.

Il y a deux techniques principalement utilisées pour déterminer la longueur d'une chaîne de caractères: 1) utiliser le caractère «null» comme caractère terminal, ou 2) la longueur est stockée en tête de la chaîne. La technique du caractère «null», telle qu'elle est utilisée par le langage C, peut ne pas être pratique quand on a besoin du caractère «null» dans les chaînes, mais elle présente l'avantage de permettre le stockage de chaînes de longueur infinie. Le stockage de la longueur dans un octet ou dans un mot limite la longueur maximale de la chaîne à 255 ou 65535 caractères respectivement, mais elle permet d'inclure le caractère «null» dans les chaînes de caractères.

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 61131-3, 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 program organization unit (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 may 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 1.5.1 a) and h) of IEC 61131-3.

4.2.2 Character strings

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

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

4.2.3 Données de type temps

Le stockage et la longueur des données de type temps dépendent de l'installation. Cela peut poser un problème de non-portabilité des fonctions et des blocs fonctionnels utilisant des données de type temps. La CEI 61131-3 prévoit donc que la plage et la précision des valeurs de type temps soient clairement spécifiées par l'implémenteur.

Le type de données TIME (durée), par exemple, peut être représenté par un nombre entier double de 32 bits sans signe, la durée étant comptée en millisecondes. Cela permet d'avoir un type de données TIME qui fournit avec précision des durées depuis une milliseconde jusqu'à 49 jours. Cela ne permet cependant pas de calculer de petites différences de temps (moins d'une milliseconde) entre des événements, ni de manipuler des différences de temps qui pourraient être négatives.

L'usage de nombres à virgule flottante pour la représentation du temps est déconseillé à cause du manque de précision dans la partie fractionnelle de la valeur. On peut, par exemple, représenter correctement sur un nombre entier double de 32 bits sans signe une durée de 30 jours, 10 minutes et 200 millisecondes c'est-à-dire T#30d10m200ms, ce que l'on ne peut pas faire avec un nombre à virgule flottante de 32 bits.

Il peut être pratique de stocker DATE, TIME_OF_DAY et DATE_AND_TIME dans le format UNIX dans lequel DATE est stocké dans un nombre entier double de 32 bits sans signe en tant que nombre de secondes depuis minuit, 1er janvier 1970, et TIME_OF_DAY est stocké en tant que nombre de secondes à partir de minuit. Cela donne la date et l'heure jusqu'à 2106 mais ne permet pas de tampon horaire pour un événement d'une précision supérieure à une seconde.

A cause des limitations de la conception de certains PSE (environnement de support de programmation) un implémenteur peut envisager le remplacement de certaines fonctions surchargées de données de type temps spécifiées en 2.5.1.5.6 de la CEI 61131-3 par des fonctions spéciales comme ADD_DATE_AND_TIME_TIME pour ajouter des variables de type TIME (durée) et DATE_AND_TIME (date et heure) et obtenir un résultat DATE_AND_TIME (date et heure). Un tel usage ne serait cependant pas conforme à 1.5.1 a) de la CEI 61131-3.

4.2.4 Variables multi-éléments

Les mises en oeuvre de la CEI 61131-3 devront limiter le nombre et la taille des dimensions des tableaux pour s'accommoder des limites des performances et de la taille mémoire des automates programmables. Une limite raisonnable pour la plupart des applications est de limiter les variables tableaux à trois dimensions.

Il faut aussi limiter le niveau d'imbrication dans l'indexation des tableaux, à cause de l'augmentation de la complexité qui intervient dans la résolution du calcul d'adresses dans l'automate programmable, plus particulièrement quand les variables utilisent des types de données dérivés complexes. Ci-dessous un exemple d'indexation très fortement imbriquée:

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

4.3 Exécution des fonctions et des blocs fonctionnels

Le présent paragraphe fournit des directives générales pour l'exécution des fonctions et des blocs fonctionnels. Quand les corps des fonctions et des blocs fonctionnels définis par l'utilisateur sont en plus programmés à l'aide de l'un des langages graphiques (LD ou FBD) définis à l'article 4 de la CEI 61131-3, la mise en oeuvre s'assurera que l'exécution obéit aux règles d'évaluation des réseaux de 4.1.3, 4.2.6 et 4.3.3 de la CEI 61131-3, ainsi qu'aux règles d'évaluation des éléments LD fournies en 4.2.2 à 4.2.5 de la CEI 61131-3.

4.2.3 Time data types

The storage and length of 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 one millisecond to 49 days. However, this would not allow the computation of small time differences (less than one millisecond) 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 minutes and 200 milliseconds, i.e. T#30d10m200ms 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 DATE is held in a 32 bit unsigned double integer as the number of seconds from midnight, January 1, 1970 and TIME_OF_DAY is held as the number of seconds from midnight. This gives dates and times of days up to the year 2106. However, this format does not allow events to be time-stamped with a precision better than one second.

Due to limitations in the design of some programming support environments (PSEs), an implementor might consider replacing some of the overloaded functions of time data types specified in 2.5.1.5.6 of IEC 61131-3 with special functions such as ADD_DATE_AND_TIME_TIME to add DATE_AND_TIME and TIME type variables to produce a DATE_AND_TIME result. However, such usage would be non-compliant according to 1.5.1 a) of IEC 61131-3.

4.2.4 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 subclause provides general guidelines for the execution of functions and function blocks. In addition, when the bodies or 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 Fonctions

On définit une fonction comme une unité d'organisation de programmes (POU) qui, quand elle est exécutée, ne génère qu'un seul élément de données (qui peut être à multi-valeurs). Une donnée interne de fonction est initialisée dynamiquement pour chaque activation, l'invocation d'une fonction avec les mêmes arguments (paramètres en entrée) doit donc toujours générer la même valeur (sortie).

NOTE Certaines fonctions normalisées définies en 2.5.1.5 de la CEI 61131-3 peuvent être *extensibles,* c'est-à-dire qu'elles peuvent avoir un nombre variable d'entrées.

Une invocation de fonction établit une liste des paramètres réels correspondant à la liste de paramètres formels spécifiés dans la déclaration de type de la fonction et elle provoque l'exécution du code de programme correspondant au corps de la fonction. En fonction de la mise en oeuvre, la liste des paramètres se compose soit des valeurs réelles des paramètres, soit des adresses où on trouvera les valeurs réelles des paramètres, soit d'une combinaison des deux, et elle peut être passée au code à exécuter comme une pile ou par d'autres moyens. Le résultat de l'exécution de la fonction sera aussi typiquement renvoyé sur la pile.

L'exécution d'une fonction peut être rendu conditionnel à l'aide de l'entrée EN décrite en 2.5.1.2 de la CEI 61131-3. Conformément aux règles 1) et 2) de 2.5.1.2 de la CEI 61131-3, l'action initiale d'un système relative à l'exécution d'une fonction est de copier l'entrée EN dans la sortie ENO. Cette variable agit comme un flux d'énergie à travers la fonction.

S'il se produit une erreur pendant le traitement de la fonction, l'action minimale exigée du système est de remettre à FALSE (faux) la sortie ENO. En fonction de la mise en oeuvre, une condition d'erreur peut aussi déclencher l'exécution d'une *tâche erreur*, définie soit par le système, soit par l'utilisateur, à la fin de l'exécution de la fonction. L'utilisateur peut associer à cette tâche un programme spécial de traitement des erreurs.

NOTE Les entrées EN et les sorties ENO des fonctions ne font pas partie de l'invocation de la fonction dans les langages ST (littéral structuré) et IL (liste d'instructions), on peut cependant lire ces variables et on peut écrire ENO en fonction de la logique des programmes écrits dans ces langages. Dans une mise en oeuvre donnée, les effets de ENO à la fin de l'exécution d'une fonction devraient être les mêmes pour tous les langages.

La lecture de l'entrée EN dans le corps d'une fonction donnera toujours la valeur booléenne TRUE (vrai), étant donné que la fonction ne sera pas évaluée quand EN est FALSE.

4.3.2 Blocs fonctionnels

Dans la CEI 61131-3, on définit un *bloc fonctionnel* comme une unité d'organisation de programmes (POU) qui, quand il est exécuté, génère une ou plusieurs valeurs en sortie; un bloc fonctionnel peut en outre avoir plusieurs instances, chacune avec ses propres données privées. Les données internes des blocs fonctionnels persistent d'une exécution à la suivante; en conséquence, des invocations successives d'un bloc fonctionnel peuvent générer des résultats différents avec les mêmes arguments (paramètres en entrée).

La CEI 61131-3 définit un certain nombre de blocs fonctionnels normalisés. Les blocs fonctionnels compteurs et minuteurs sont typiquement installés dans le «firmware» des automates.

Une *invocation* de bloc fonctionnel établit les valeurs des variables en entrée du bloc fonctionnel et provoque l'exécution du code de programme correspondant au corps du bloc fonctionnel. Ces valeurs peuvent être établies graphiquement en connectant des variables ou des sorties d'autres fonctions ou d'autres blocs fonctionnels aux entrées correspondantes, ou littéralement en inscrivant les valeurs affectées dans les variables en entrée. Si une variable n'a pas reçu de valeur lors de l'invocation du bloc fonctionnel, on utilise une valeur par défaut. En fonction de la mise en oeuvre, les variables en entrée peuvent être les valeurs réelles des paramètres, ou les adresses où l'on trouvera ces valeurs réelles, ou une combinaison des deux. Ces valeurs sont toujours passées au code exécutant dans la structure de données

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

NOTE 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 parameters corresponding to the list of formal parameters 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 parameters may consist of the actual parameter values, of the addresses at which to locate the actual parameter 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 the 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 The EN input and ENO output of functions are not part of function invocations in the structured text (ST) and instruction list (IL) languages; however, these variables can be read and ENO can be written, from program logic written in these languages. The effect of ENO at the end of function execution should be the same for all languages in a given implementation.

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 parameters).

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 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 parameter values of the addresses at which to locate the actual parameter values or a combination of the two. These values are always passed to the executing code in the data

associée à l'instance du bloc fonctionnel. Les résultats de l'exécution du bloc fonctionnel sont également renvoyés dans cette structure de données. Lorsque l'invocation d'un bloc fonctionnel est mise en oeuvre comme un appel de procédure, il suffit de passer un seul argument à la procédure pour l'exécution: l'adresse de la structure des données de l'instance.

NOTE 1 Quand une instance de bloc fonctionnel dans un programme est associée à une tâche séparée (voir tableau 50, points 3a et 3b de la CEI 61131-3), il convient que l'invocation du bloc fonctionnel par le programme établisse des valeurs pour les variables en entrée du bloc fonctionnel, mais ne provoque pas l'exécution du code programme associé au corps du bloc fonctionnel. L'exécution de ce code sera exclusivement commandée à partir de la tâche associée selon la règle 5) de 2.7.2 de la CEI 61131-3.

NOTE 2 Quand une instance de bloc fonctionnel sert d'interface à du matériel à grande vitesse, tel que des compteurs ou des convertisseurs A/D rapides, ou quand l'instance du bloc fonctionnel est exécutée de façon préemptive par une tâche périodique à grande vitesse ou menée par des interruptions, les valeurs de sortie réelles du bloc fonctionnel peuvent changer pendant que les traitements concernés par ces sorties sont exécutés dans le programme contenant l'instance du bloc fonctionnel. Dans ce cas, il faut que la mise en oeuvre fournisse le moyen de geler les sorties pendant que ces traitements ont lieu, par exemple en fournissant un tampon temporaire aux valeurs de sortie réelles.

Des types de données élémentaires supplémentaires R_EDGE et F_EDGE sont disponibles seulement pour les variables en entrée des blocs fonctionnels. Ces types de données fournissent respectivement une détection de front montant (0 --> 1) ou descendant (1 --> 0) d'entrées booléennes lors de l'invocation d'un bloc fonctionnel. La variable condition n'est à TRUE que quand le front spécifié est détecté, autrement elle est à FALSE.

Contrairement à l'exécution d'une fonction, l'exécution d'un corps de bloc fonctionnel ne se réduit pas à la valeur TRUE de l'entrée EN. Cependant, l'exécution de fonctions dans le corps du bloc fonctionnel aura le même effet sur le traitement des erreurs que celui décrit dans le paragraphe précédent.

4.4 Mise en oeuvre des SFC (diagramme fonctionnel en séquence)

Un certain nombre de points relatifs à la mise en oeuvre de SFC ont déjà été discutés dans le présent rapport. Ces points sont les suivants.

1) La commande des actions peut être mise en oeuvre soit sous la forme d'un bloc de commande d'action, soit de son équivalent fonctionnel dans un code et des structures de données optimisés: voir 3.9.1 ci-dessus et 2.6.4.5 de la CEI 61131-3.
2) Il faut utiliser une méthode cohérente, telle que l'algorithme à quatre étapes décrit en 3.9.4, pour l'exécution des évolutions SFC, que le SFC survienne dans un programme, dans un bloc fonctionnel ou dans une action SFC.
3) Il est recommandé que les variables «indicateur» décrites en 2.6.4.3 de la CEI 61131-3 soient supportées en tant que facilité de notation et sans aucune fonctionnalité spécifique dépendant de la mise en oeuvre, afin de maximiser la souplesse et la portabilité dans l'utilisation de ces dispositifs, tels qu'ils sont discutés en 3.9.6.
4) Il peut être utile dans une installation d'imposer la discipline de programmation en SFC décrite en 3.12.4.
5) Comme on le note en 4.1, lors de l'allocation de stockage pour un programme ou un bloc fonctionnel contenant des SFC, il est nécessaire d'envisager les besoins de stockage des informations d'état SFC ainsi que les variables utilisées par le programme ou le bloc fonctionnel. Les informations d'état SFC comprennent les drapeaux de phase et les blocs de commande d'actions (éventuellement), le temps utilisé par l'étape (s'il est supporté), et d'autres informations d'état exigées par la mise en oeuvre.

4.5 Ordonnancement des tâches

La structure TASK est définie en 2.7 de la CEI 61131-3 pour permettre aux utilisateurs de spécifier les exigences d'ordonnancement de l'exécution des programmes et des blocs fonctionnels sans avoir à développer à la main un cycle d'exécution détaillé. Ces exigences peuvent être combinées à des techniques modernes d'ordonnancement pour des systèmes en temps réel pour déterminer à l'avance si le système peut être ordonné de façon à satisfaire

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 (see 3a and 3b) of table 50 of IEC 61131-3), 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 are processes 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, e.g. by providing a temporary buffer for the actual output value.

Additional elementary data types 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.

In contrast to function execution, the execution of a function block body is not restricted to the TRUE condition of an EN input. However, the execution of functions in the function block body will have the same effects on error processing as described in the preceding subclause.

4.4 Implementation of sequential function charts (SFCs)

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

1) Action control may be implemented either as an action control function block or its functional equivalent in optimized code and data structures; see 3.9.1 and 2.6.4.5 of IEC 61131-3.
2) A consistent method, such as the four step algorithm described in 3.9.4, should be utilized for execution of SFC evolution, whether the SFC occurs in programs, function blocks or SFC actions.
3) 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.
4) It may be useful for an implementation to enforce the SFC programming disciplines described in 3.12.4.
5) As noted in 4.1, 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 (if any), step elapsed times (if supported) and other state information as required by the implementation.

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 function blocks, 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 [2], especially in systems where

aux exigences de l'utilisateur [2], plus spécialement dans les systèmes où on utilise l'ordonnancement préemptif (voir 3.10.2 pour une discussion d'ordonnancement préemptif versus non préemptif). Il convient que les implémenteurs soient avertis de ces techniques et envisagent leur mise en oeuvre dans les systèmes conformes à la CEI 61131-3.

4.5.1 Classification des tâches

Quand une tâche est déclenchée, elle planifie l'exécution des programmes et des blocs fonctionnels associés. On peut, par conséquent, caractériser les tâches de la CEI 61131-3 par leur mécanisme de déclenchement:

– une tâche périodique est déclenchée régulièrement à des intervalles de temps déterminés. Cela est configuré par l'utilisateur en connectant l'entrée SINGLE du bloc de la tâche à la valeur booléenne FALSE (on en la laissant déconnectée), et en donnant à l'entrée INTERVAL une valeur non nulle de type TIME, représentant l'intervalle de déclenchement;

– une tâche non périodique est déclenchée par un événement intérieur ou extérieur qui ne survient pas nécessairement à intervalles réguliers. Cela est configuré par l'utilisateur en connectant l'entrée SINGLE du bloc de la tâche à une variable booléenne dont le front montant représente l'événement de déclenchement, et par la mise de l'entrée INTERVAL à t#0s (ou en la laissant déconnectée);

– la tâche par défaut est automatiquement associée aux programmes qui ne sont pas explicitement associés à des tâches et elle planifie les programmes associés à la manière des essais circulaires avec la priorité du système la plus basse. Cette tâche peut aussi gérer le balayage des entrées et des sorties comme indiqué en 2.1.

Les tâches non périodiques peuvent être classées selon la nature de l'événement de déclenchement. Ces événements peuvent être:

– «départ à froid» ou «départ à chaud», définis en 2.4.2 de la CEI 61131-3;

– erreur pendant l'exécution, définie en 1.5.1 d)4) de la CEI 61131-3 et énumérée à l'annexe E de la CEI 61131-3;

– des événements détectés ou générés par des mécanismes matériels ou logiciels dépendant de l'installation (appelés quelquefois interruptions), tels que les fronts montants d'un signal électrique ou la fin du décompte d'un compteur matériel rapide.

Il convient que l'implémenteur envisage la spécification des noms des variables booléennes ou des variables booléennes représentées directement (comme décrit en 2.4.1.1 de la CEI 61131-3) dont le front montant représente des événements tels que ceux décrits plus haut pour un type particulier de ressources ou d'instances défini en 2.7.1 de la CEI 61131-3. Cela simplifiera l'association de ces événements avec des tâches qui peuvent planifier les programmes pour répondre à l'occurrence de ces événements. Les implémenteurs envisageront également la livraison de programmes par défaut pour traiter des types d'événements les plus fréquents tels que les redémarrages et les erreurs pendant l'exécution.

4.5.2 Priorités des tâches

Les interactions entre l'assignation de priorités à des tâches, les intervalles d'ordonnancement de tâches périodiques, les débits d'arrivées des tâches non périodiques et les temps d'exécution des tâches peuvent avoir des effets majeurs sur les temps de réponses et la possibilité d'ordonnancement des systèmes en temps réel [2]. Il convient que les implémenteurs de systèmes conformes à la CEI 61131-3 fournissent des outils d'ordonnancement adéquats pour les dispositifs des tâches supportés par l'installation.

preemptive scheduling is supported (see 3.10.2 for a discussion of preemptive vs. 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 function blocks. 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.

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 1.5.1 d) 4) 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 [2]. Implementors of IEC 61131-3 compliant systems should provide adequate scheduling tools and facilities for the tasking features supported by the implementation.

4.6 Traitement des erreurs

4.6.1 Mécanismes de traitement des erreurs

Les paragraphes 1.5.1 c) et d) de la CEI 61131-3 spécifient le traitement des erreurs dans les systèmes conformes.

Le point c) s'applique typiquement aux erreurs de syntaxe ou de configuration dans le programme source. Ces erreurs peuvent être détectées au moment où elles sont entrées dans le PSE (environnement de support de programmation) par l'utilisateur, pendant l'analyse du programme dans l'étape de compilation (le cas échéant), pendant l'édition de liens de la POU (unité d'organisation de programmes) dans une configuration ou pendant le chargement du logiciel configuré dans l'automate pour exécution.

Le point d) se rapporte aux erreurs énumérées à l'annexe E de la CEI 61131-3, qui sont pour la plupart des erreurs qui peuvent survenir pendant l'exécution d'un programme utilisateur, à savoir des «erreurs d'exécution». Le point d) donne quatre possibilités pour traiter ces erreurs:

1) la documentations d'accompagnement avertit que l'erreur n'est pas signalée;
2) le système avertit, pendant la préparation du programme pour l'exécution, que cette erreur peut éventuellement se produire;
3) le système signale l'erreur pendant la préparation du programme pour exécution;
4) le système signale l'erreur pendant l'exécution du programme et initialise les procédures appropriées de traitement d'erreurs, système ou utilisateur.

NOTE 1 L'utilisation de l'option 1) n'est pas recommandée.

NOTE 2 L'option 3) est typiquement mutuellement exclusive avec les options 2) et 4). Les options 2) et 4) ne sont cependant pas mutuellement exclusives et il convient de les utiliser en combinaison chaque fois que c'est possible. C'est-à-dire que, quand l'erreur ne peut pas être détectée avant l'exécution par l'option 3), l'utilisateur est prévenu que cette erreur peut se produire et qu'elle sera alors détectée pendant l'exécution.

NOTE 3 Les points f) et g) en 1.5.1 de la CEI 61131-3 exigent que les extensions et les dispositifs dépendants de la mise en oeuvre soient traités de la même façon que les erreurs. L'implémenteur peut cependant fournir un commutateur logiciel qui permet à l'utilisateur d'éviter ce traitement.

Le tableau 3 recommande les mécanismes de traitement d'erreurs à appliquer aux conditions énumérées au tableau E.1 de la CEI 61131-3.

4.6 Error handling

4.6.1 Error handling mechanisms

Subclauses 1.5.1 c) and d) 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 Programming Support Environment (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) 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 the 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.

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) 1.5.1 of IEC 61131-3 require that extensions and implementation-dependent features be processed in the same manner as errors. However, the implementor may supply a software switch by means of which the user can disable such processing.

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

Tableau 3 – Mécanismes de traitement d'erreurs recommandés

Paragraphe	**Conditions d'erreurs** (NOTE 1)	**Mécanismes** (Note 2)
2.3.3.1	Valeur de la variable hors de l'intervalle spécifié	RT
2.4.2	La longueur de la liste d'initialisation ne correspond pas au nombre d'entrées dans le tableau	ED
2.5.1.5.1	Erreur de conversion de type	ED
2.5.1.5.2	Résultat numérique hors des limites pour le type de données Division par zéro	RT ED, RT
2.5.1.5.4	Mélange de types de données en entrée d'une fonction de sélection Sélecteur (K) hors de la plage pour la fonction MUX	ED RT
2.5.1.5.5	Position invalide du caractère spécifié Résultat dépassant la longueur maximale d'une chaîne	EW, RT
2.5.1.5.6	Résultat hors de la plage pour le type de données	RT
2.5.2.2 (NOTE 2)	Pas de valeur de paramètre spécifiée pour une instance de bloc fonctionnel utilisée comme paramètre d'entrée	ED
2.5.2.2 (NOTE 3)	Pas de valeur de paramètre spécifiée pour le paramètre VAR_IN_OUT	ED
2.6.2	Pas, ou plus d'une, étape initiale dans un réseau SFC Programme utilisateur cherchant à modifier l'état ou l'heure de l'étape	ED
2.6.2.5	Transitions simultanément vraies et non prioritaires dans une divergence de sélection	ED, EW
2.6.3	Effets indésirables dans l'évaluation d'une condition de transition	ED
2.6.4.5	Erreur d'encombrement de commande d'action	EW, RT
2.6.5	Schéma diagramme fonctionnel détruit ou inaccessible	ED
2.7.1	Conflit de type de données dans VAR_ACCESS	ED
2.7.2	Tâches demandant trop de temps processeur Délais d'exécution non respectés Autres conflits d'ordonnancement de tâches	ED ED EW, RT
3.2.2	Résultat numérique hors de la plage pour le type de données	EW, RT
3.3.1	Division par zéro Type de données invalide pour l'opération	ED, EW, RT ED
3.3.2.1	Retour d'une fonction sans valeur assignée	ED
3.3.2.4	Boucle sans fin	ED, EW, RT
4.1.1	Identificateur servant à la fois d'étiquette de connecteur et de nom d'élément	ED
4.1.5	Comme pour 2.5.1.5.2	ED, RT

NOTE 1 Le présent tableau n'est pas une liste exhaustive de toutes les erreurs d'exécution possibles. Les implémenteurs peuvent élargir ce tableau ainsi que les facilités de traitement d'erreurs correspondantes.

NOTE 2 ED = détection préventive (early detection), 1.5.1 d)3) de la CEI 61131-3

EW = avertissement préventif (early warning), 1.5.1 d)2) de la CEI 61131-3

RT = détection pendant l'exécution (run time detection), 1.5.1 d)4) de la CEI 61131-3

NOTE 3 Ces erreurs sont traitées dans le corrigendum à la CEI 61131-3.

Table 3 – Recommended error handling mechanisms

Subclause	Error conditions (NOTE 1)	Mechanisms (NOTE 2)
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
2.5.1.5.1	Type conversion errors	ED
2.5.1.5.2	Numerical result exceeds range for data type	RT
	Division by zero	ED,RT
2.5.1.5.4	Mixed input data types to a selection function	ED
	Selector (K) out of range for MUX function	RT
2.5.1.5.5	Invalid character position specified	EW,RT
	Result exceeds maximum string length	
2.5.1.5.6	Result exceeds range for data type	RT
2.5.2.2 (NOTE 2)	No parameter value specified for a function block instance used as input parameter	ED
2.5.2.2 (NOTE 3)	No parameter value specified for a VAR_IN_OUT parameter	ED
2.6.2	Zero or more than one initial step in SFC network	ED
	User program attempts to modify step state or time	
2.6.2.5	Simultaneously true, non-prioritized transitions in a selection divergence	ED,EW
2.6.3	Side effects in evaluation of transition condition	ED
2.6.4.5	Action control contention error	EW,RT
2.6.5	"Unsafe" or "unreachable" SFC	ED
2.7.1	Data type conflict in VAR_ACCESS	ED
2.7.2	Tasks require too many processor resources	ED
	Execution deadline not met	ED
	Other task scheduling conflicts	EW,RT
3.2.2	Numerical result exceeds range for data type	EW,RT
3.3.1	Division by zero	ED,EW,RT
	Invalid data type for operation	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
4.1.5	As for 2.5.1.5.2	ED,RT

NOTE 1 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 2 ED = early detection as per 1.5.1 d)3) of IEC 61131-3

EW = early warning as per 1.5.1 d)2) of IEC 61131-3

RT = run time detection as per 1.5.1 d)4) of IEC 61131-3

NOTE 3 These errors will be dealt with in the corrigendum to IEC 61131-3.

4.6.2 Procédures de traitement des erreurs d'exécution

Ce paragraphe contient des recommandations relatives à la mise en oeuvre de 1.5.1 d)4) de la CEI 61131-3 ...«Le système doit signaler les erreurs survenues pendant l'exécution d'un programme et initialiser les procédures système ou celles définies par l'utilisateur de traitement des erreurs...»

NOTE Le domaine de cette disposition de la CEI 61131-3 se limite aux erreurs dans les programmes de l'utilisateur; les implémenteurs peuvent cependant envisager de fournir des procédures similaires pour le traitement des erreurs provenant d'autres sources, telles que les erreurs des sous-systèmes d'entrées/sorties ou de communications.

4.6.2.1 Comptes rendus d'erreurs

L'information dont il faut rendre compte lorsque survient une erreur contiendra:

– la notification du fait qu'une erreur est survenue;
– la classification du type de l'erreur (par exemple «division par zéro»);
– l'identification de la source de l'erreur (par exemple une POU).

Afin de d'obtenir un style uniforme de compte rendu d'erreurs, l'implémenteur envisagera de coder cette information dans des variables d'un même type comme suit:

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

NOTE 1 La longueur des éléments STRING ci-dessus dépendra de la mise en oeuvre.

NOTE 2 L'utilisation des types nombres entiers ou énumérations pour les classes d'éléments d'erreurs peut être envisagée quand on a besoin d'une plus grande efficacité dans le traitement subséquent de l'erreur.

Les variables de compte rendu d'erreur seront typiquement mises à disposition en tant que déclarations par défaut spécifiques à la mise en oeuvre, par exemple en tant que déclarations de variables globales pour un type spécifique de ressource, ou en tant que variables de sortie des programmes ou des tâches. Un ensemble de déclarations d'erreurs pour une ressource pourrait être, par exemple:

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

NOTE 1 Dans cet exemple, le commentaire se réfère aux paragraphes de la CEI 61131-3 où sont décrits les points qui pourraient provoquer l'erreur particulière.

NOTE 2 On peut utiliser une variable unique de compte rendu d'erreur, mais on peut obtenir une meilleure performance dans le traitement des erreurs en utilisant un plus grand nombre de variables, afin de produire une résolution plus fine dans la classification des erreurs.

4.6.2 Run-time error handling procedures

This subclause contains recommendations for the implementation of 1.5.1 d) 4) of IEC 61131-3, ..."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 (e.g. "division by zero");
- identification of the source of the error (e.g. 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 use of integer or enumerated types for the error 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, e.g. 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 item 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 Procédures de traitement des erreurs définies dans le système

Le traitement des erreurs d'exécution comporte en général les étapes suivantes:

1) le flux normal d'exécution du programme est suspendu;
2) on exécute l'action appropriée au type d'erreur, par exemple:
   - l'erreur peut être corrigée quand cela est possible,
   - si la correction n'est pas possible, on peut substituer une valeur par défaut à la variable erronée,
   - on peut rendre compte de l'occurrence d'une erreur et de toute correction effectuée à un opérateur ou cette correction peut être enregistrée dans un fichier pour référence ultérieure;
3) on reprend l'exécution du programme à l'endroit approprié. En fonction du type de l'erreur et des possibilités d'actions correctives, ces reprises peuvent être immédiates ou soumises à un changement d'état du système tel qu'un redémarrage à chaud, un redémarrage à froid ou une commande venant du réseau de communications ou d'un opérateur.

Il convient que l'implémenteur spécifie, pour chaque type d'erreur d'exécution traité par le système, les actions correctives effectuées, les comptes rendus faits et les procédures de reprises appliquées. Si le compte rendu des erreurs est modélisé par des variables globales, comme indiqué dans le paragraphe précédent, la gestion des erreurs peut être modélisée par un ordonnancement préemptif ou des tâches de traitement des erreurs définis en 2.7.2 de la CEI 61131-3. Dans l'exemple donné dans le paragraphe précédent, la spécification spécifique à la ressource des tâches de traitement des erreurs pourrait prendre la forme:

```
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 L'implémenteur spécifiera les niveaux de priorité pour le traitement des diverses erreurs. Ces priorités sont habituellement supérieures à celles des tâches du programme utilisateur, afin de garantir que ce programme sera interrompu pour le traitement de l'erreur.

NOTE 2 L'occurrence et le traitement des erreurs peut avoir un impact sévère sur la capacité du système à respecter ses limites d'ordonnancement.

Avec le modèle ci-dessus, la correction des erreurs et les procédures de compte rendu peuvent être spécifiées en associant les tâches de traitement des erreurs à des programmes appropriés de traitement des erreurs définis dans le système. Dans l'exemple ci-dessus, les déclarations de ces associations pourraient être:

```
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;
```

4.6.2.2 System-defined error handling procedures

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

1) the normal flow of program execution is suspended;
2) the 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;
3) 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 or 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.

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;
```

L'implémenteur pourrait alors spécifier les actions à entreprendre par le programme spécifié de traitement des erreurs, pour chaque classification possible d'erreurs dans la variable globale ERROR_REPORT, par exemple, les actions à entreprendre par le programme SYSTEM_MATH_ERROR_PROCEDURE pour chacune des valeurs possibles de MATH_ ERROR.CLASS, etc.

4.6.2.3 Procédures utilisateurs de traitement des erreurs

Une mise en oeuvre selon la CEI 61131-3 peut fournir les moyens pour des procédures utilisateurs de traitement des erreurs. Dans les exemples du paragraphe précédent, on peut réaliser cela en substituant ou en étendant les programmes systèmes de traitement des erreurs par des programmes utilisateurs associés aux tâches de traitement des erreurs appropriées.

Si les procédures utilisateurs de traitement des erreurs sont supportées, l'implémenteur fournira des facilités pour ces procédures pour spécifier les mêmes types de mécanismes de traitements et de comptes rendus d'erreurs, ainsi que les options de reprise du programme utilisateur, de la même manière que les procédures système. Ces facilités peuvent prendre la forme, par exemple, de variables globales dont la valeur peut être fixée, ou de blocs fonctionnels qui peuvent être invoqués, par les procédures de traitement des erreurs définies par les utilisateurs.

4.7 Interface système

Il convient que les implémenteurs envisagent, dans les ressources, une réserve de variables globales (qui pourrait inclure des instances de blocs fonctionnels), pour des besoins d'interfaces avec les fonctions système. Par exemple, comme décrit en 4.5.1, des variables booléennes globales peuvent servir à représenter des états ou des événements spécifiés par le système, comme BATTERY_LOW (accumulateur déchargé) ou POWER_ABOUT_TO_FAIL (proche de la panne d'alimentation en énergie). On peut aussi représenter les entités systèmes comme des instances de blocs fonctionnels spécifiques du système, par exemple BATTERY (accumulateur) ou POWER_SUPPLY (alimentation en énergie), avec des variables d'entrée et de sortie définies, représentant leur état ou leurs interfaces de commande.

4.8 Conformité

La CEI 61131-3 contient de nombreuses exigences de conformité en plus des exigences générales énumérées en 1.5.1 de la CEI 61131-3. Les implémenteurs feront attention à toute utilisation de l'expression «il faut» dans la CEI 61131-3, car chacune d'elles indique une exigence (de conformité).

Les paragraphes suivants traitent d'un nombre de dispositions plus importantes que les implémenteurs devraient garder en mémoire pour le développement des systèmes d'implémentation conformes à la CEI 61131-3.

4.8.1 Déclaration de conformité

La première exigence énumérée en 1.5.1 de la CEI 61131-3 est qu'une déclaration de conformité doit être incluse dans la documentation ou produite par le système lui-même, par exemple sous la forme d'un fichier que l'on peut lire ou imprimer. Cette déclaration se compose d'une déclaration de conformité et d'une série de tableaux énumérant les dispositifs supportés par le système; le format exact de ces tableaux est prescrit en 1.5.1 de la CEI 61131-3.

NOTE La déclaration de conformité n'est pas le seul élément de documentation imposé par la CEI 61131-3; voir, par exemple, 1.5.1b), d)1), e) et i) de la CEI 61131-3.

Etant donné la grande variété d'applications des automates programmables, ce format des déclarations de conformité a été jugé plus pratique que l'énumération de classes de conformité. L'expérience acquise lors de l'utilisation des éléments de la CEI 61131-3 rendra peut-être possible, dans les révisions futures, la définition de classes de conformité.

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, e.g. 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 function. For instance, as described in 4.5.1, global Boolean variables may be used to represent system-specified status or events, e.g. BATTERY_LOW or POWER_ABOUT_TO_FAIL. Alternately, system entities may be represented as instances of system-specific function blocks, e.g. 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, e.g. 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, 1.5.1b), d) 1), e) and i) of IEC 61131-3.

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 Jeux d'instructions d'automates

Le paragraphe 1.1 de la CEI 61131-3 limite spécifiquement son domaine à une «représentation imprimée et affichée ... des langages de programmation à utiliser pour les automates programmables...»; on n'exige pas, en particulier, que le langage IL (liste d'instruction) défini en 3.2 de la CEI 61131-3 soit considéré comme un jeu d'instructions pour une machine réelle ou virtuelle. Le langage IL est plutôt considéré comme un moyen d'exprimer la plupart des fonctionnalités disponibles dans les autres langages de la CEI 61131-3 dans un format de langage d'assemblage familier à un grand nombre d'utilisateurs des systèmes actuels.

En principe, les éléments de langage de la CEI 61131-3 peuvent être compilés pour un grand nombre de jeux d'instructions de machines. Le PSE (environnement de support de programmation) doit être capable de présenter à l'utilisateur la programmation de systèmes d'automates programmables dans les formats prescrits par la CEI 61131-3, et il convient qu'il cache à l'utilisateur les détails des jeux d'instructions des machines. Cela permettra aux architectures des automates programmables d'évoluer tout en protégeant les investissements des utilisateurs en logiciel et en formation.

Une des conséquences naturelles de cette souplesse vis-à-vis des jeux d'instructions est que les programmes conformes, définis en 1.5.2 de la CEI 61131-3, ne seront portables qu'au niveau du code source.

4.8.3 Essais de conformité

Le développement et l'exécution des suites d'essais de conformité pour les langages de la CEI 61131-3 dépendront de l'acquis d'expériences pratiques dans leur application; ils feront l'objet d'une future normalisation.

4.9 Compatibilité avec la CEI 60617-12, la CEI 60617-13 et la CEI 60848

Les éléments de la CEI 61131-3 ont été définis de façon à être aussi compatibles que possible avec la CEI 60617-12, la CEI 60617-13 et la CEI 60848. Dans le même temps, la CEI 61131-3 devrait être aussi cohérente que possible avec les pratiques de programmation des automates programmables existantes, et elle était soumise aux restrictions des jeux de caractères de l'ISO/CEI 646.

Il convient que les implémenteurs de systèmes conformes à la CEI 61131-3 avec des représentations graphiques et semi-graphiques de programmes envisagent l'extension de ces représentations afin d'obtenir une meilleure compatibilité avec les représentations graphiques de la CEI 60617 et de la CEI 60848.

5 Exigences pour PSE (environnement de support de programmation)

5.1 Interface utilisateur

L'interface utilisateur est le moyen principal pour rendre visibles à l'utilisateur les dispositifs et les fonctionnalités d'un système de commande programmable, et elle est souvent la base fondamentale de l'opinion de l'utilisateur sur le système. Le développement de l'interface utilisateur est donc un problème majeur pour les créateurs de logiciels et de matériels d'automates programmables.

Il y a évidemment différents moyens de conception et de mise en oeuvre des fonctions utilisateur, même quand on utilise la même norme consacrée au langage de programmation. Une compréhension de la façon dont il et prévu, dans la norme, d'utiliser les éléments du langage conduira cependant à une conception et une mise en oeuvre plus satisfaisante du PSE.

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 instruction list (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 as 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. The programming support environment (PSE) 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 whilst 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.

4.9 Compatibility with IEC 60617-12, IEC 60617-13 and IEC 60848

The elements of IEC 61131-3 were established to be as compatible as possible with IEC 60617-12, IEC 60617-13 and with IEC 60848. At the same time, IEC 61131-3 had to be consistent as possible with existing practice for programming of programmable controllers and was subject to the restrictions of the ISO/CEI 646 character set.

Implementors of IEC 61131-3 compliant systems with graphic and semigraphic representation of programs should consider extending such representations to achieve greater compatibility with the graphic representations to achieve greater compatibility with the graphic representations in IEC 60617 and IEC 60848.

5 Programming support environment (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 a more satisfactory PSE design and implementation.

Pour tout produit logiciel, il est critique de savoir quelles sont les informations utiles, quand elles sont utiles et comment il faut les afficher. La conception du PSE doit respecter les principes ergonomiques fondamentaux, par exemple:

– plus le PSE est complexe, plus il est nécessaire d'informer et de guider l'utilisateur sur son utilisation;
– il convient que le PSE guide l'utilisateur à travers les étapes d'une méthodologie systématique de développement du logiciel. Il faut, cependant, que l'utilisateur puisse parcourir ces étapes dans un ordre qu'il a déterminé;
– le PSE peut guider l'utilisateur avec des réponses appropriées sur l'écran et une organisation des fenêtres, et avec un jeu de dialogues appropriés à l'étape en cours de la méthodologie logicielle;
– quand elle est disponible, une aide contextuelle en ligne sera apportée en réponse aux questions des utilisateurs. Les informations d'aide présentées sans être demandées, sont systématiquement ignorées;
– des études récentes montrent que l'utilisateur ne veut pas une formation exhaustive et coûteuse en temps. Il convient qu'un matériel de formation en ligne soit disponible, permettant à l'utilisateur d'exécuter, après une courte période d'étude, les fonctions les plus couramment utilisées, et d'étudier les méthodes plus avancées quand il en a besoin, dans le cours normal d'un travail productif;
– l'utilisation de documents sur papier est antiéconomique à la fois pour le vendeur et pour l'utilisateur du PSE; il ne faut donc les utiliser que comme documents de référence en supplément à la documentation en ligne.

Quand on utilise un système d'interface utilisateur à multiples fenêtres, il faut suivre les directives ergonomiques du fournisseur du système de fenêtrage afin de parvenir à une forte productivité de la part de l'utilisateur et à un faible taux d'erreurs dans un minimum de temps de formation. Ces directives incluent typiquement:

– une correspondance entre les outils de pointage (souris, etc.) et le clavier;
– des conventions pour sélectionner et «désélectionner» les objets;
– des règles pour utiliser certaines fenêtres, comme les boîtes à messages, les boîtes de listes, les boîtes de dialogues, etc.;
– un ordre pour les items dans les menus.

5.2 Ecriture des programmes, des fonctions et des blocs fonctionnels

L'utilisateur doit accomplir un ensemble similaire de tâches dans l'écriture de toute unité d'organisation de programmes (POU), que ce soit une fonction, un bloc fonctionnel ou un programme. La programmation est la structuration d'une POU, que ce soit par une approche ascendante ou descendante, ou les deux. Une POU contient des déclarations (paramètres et variables locales) et un corps écrit dans n'importe quel langage. Les moyens de déclarer les paramètres et les variables peuvent être indépendants du langage dans lequel on les utilise. La CEI 61131-3 normalise les syntaxes littérales des déclarations de données, mais elle n'exige pas qu'elles soient présentées ou saisies par l'utilisateur, sous cette forme, pendant la programmation.

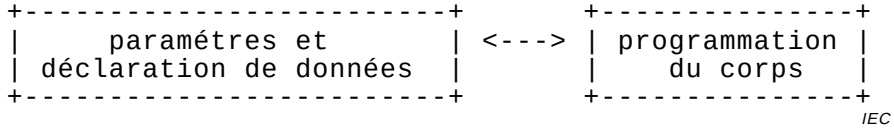


Figure 21 – Phases essentielles de la création d'une unité d'organisation de programmes (POU)

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:

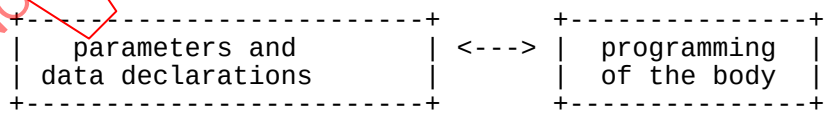
– the more complex the PSE is, the more the user has to be guided and informed about its use;
– the PSE should guide the user through the steps of a systematic software development methodology. However, it should be possible for the user to perform these steps in a user-determined order;
– the PSE can guide the user with appropriate cues on a screen and window organization and with a set of dialogs appropriate to the step of the software methodology currently being performed;
– 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 devices (mouse, etc.) and keyboard operations;
– conventions for selecting and deselecting objects;
– rules for using special windows such as message boxes, list boxes, dialog 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 program organization unit (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. A POU contains declarations (parameters and local variables) and a body programmed in any language. The means of declaring parameters and 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.



IEC 1637/99

Figure 21 – Essential phases of program organization unit (POU) creation

Il faut sérieusement envisager la mise à disposition de dispositifs tels que des éditeurs dirigés par la syntaxe, pour une détection immédiate et une (éventuelle) correction des erreurs de syntaxe pendant les déclarations et la programmation des POU. Ces dispositifs augmentent la productivité et la qualité de la programmation de n'importe quel langage. Cet avantage sera, cependant, spécialement mis en évidence dans les PSE pour les langages de la CEI 61131-3 pour les raisons suivantes:

– chaque POU peut être écrite dans un langage littéral ou graphique différent. La détection et la correction immédiates des erreurs peut aider l'utilisateur à apprendre la syntaxe d'un langage qui lui est moins familier;
– quand une POU est réalisée dans un langage graphique, il est plus facile de commettre des erreurs telles que connexions entre variables de types incompatibles. La détection et la correction immédiates de telles connexions peut empêcher la génération d'un grand nombre de messages d'erreurs gênants pendant la compilation;
– les tâches de déclaration, d'édition et de compilation d'une POU peuvent être accomplies à des moments différents, et même par des utilisateurs différents. Il est donc impératif que les erreurs syntaxiques soient détectées et évitées le plus tôt possible lors de l'activité de programmation.

Il devrait être possible, pour l'utilisateur, d'activer et de désactiver la détection automatique des erreurs de syntaxe aussi souvent qu'il le désire.

5.3 Conception et configuration des applications

La CEI 61131-3 sépare la configuration d'une application de sa programmation. Le PSE doit donc séparer distinctement ces deux phases sans leur attribuer obligatoirement un ordre. Cela veut dire qu'il doit être possible de programmer jusqu'à un certain niveau avant de procéder à une configuration.

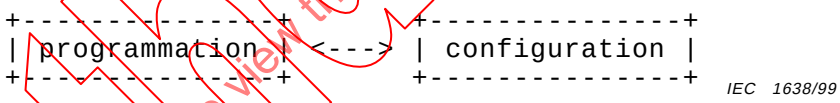


Figure 22 – Phases essentielles de la création d'une application

Comme le décrit la CEI 61131-3, le PSE doit aider l'utilisateur à réaliser la configuration du système. Cela inclut des tâches telles que la description des ressources, la déclaration des variables globales, l'assignation, par tâche, des programmes appelés, la gestion des bibliothèques de programmes, la définition des communications entre l'application et des entités externes, etc. Par exemple, pour assister l'utilisateur dans la configuration des chemins d'accès, tels qu'ils sont définis en 2.7 de la CEI 61131-3, un éditeur graphique pourrait être fourni pour permettre à l'utilisateur de spécifier les chemins d'accès dans le format de la figure 19 de la CEI 61131-3. Cet éditeur pourrait générer immédiatement les instructions VAR_ACCESS... END_VAR nécessaires pour des déclarations de type correctes sur le plan de la syntaxe et de la sémantique.

5.4 Compilation séparée

La compilation séparée d'une POU (unité d'organisation de programmes) peut être analysée en termes de dépendance. Une POU qui déclare une variable devient dépendante de la déclaration de type de la variable. Afin de simplifier les explications et les figures données ci-après, les dépendances vis-à-vis des types de données sont prises en compte mais pas explicitement décrites dans chaque cas.

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.

5.3 Application design and configuration

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.

Figure 22 – 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 program organization units (POUs) can be analyzed 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.

Le cas le plus simple à envisager comprend une POU qui ne dépend pas d'autres POU. Par exemple, une fonction qui n'appelle pas d'autres fonctions peut être compilée toute seule. De la même façon, un bloc fonctionnel dont les variables déclarées (paramètres, variables locales et externes) ne sont pas des instances de blocs fonctionnels, et dont le corps ne contient pas un appel à une autre fonction, peut être compilé tout seul. La compilation peut traiter directement à partir d'une source graphique ou littérale de la POU; dans ce cas, les avantages de séparer la compilation sont évidents:

– le balayage, l'analyse, l'analyse sémantique et les essais indépendants sont suffisants pour vérifier et valider une POU;
– la réutilisation des POU est simplifiée si le code généré translatable peut être associé à la source.

La compilation séparée des blocs fonctionnels qui utilisent des variables externes déclarées à l'aide de VAR_EXTERNAL est directe, comme l'est aussi la compilation séparée de programmes qui utilisent à la fois des variables VAR_EXTERNAL et des variables représentées directement. Il est possible de compiler ces POU et de résoudre les liens inconnus pendant la configuration de l'application.

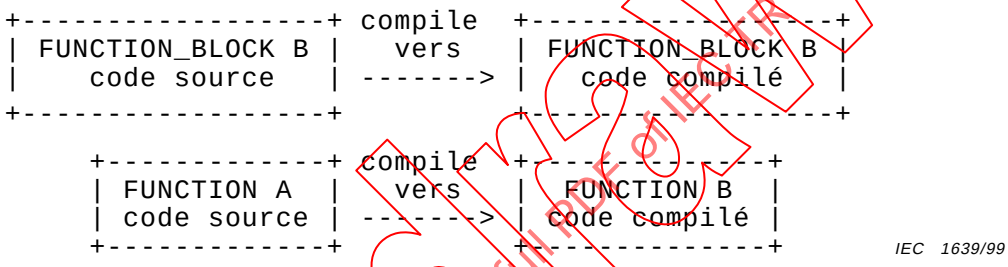


Figure 23 – Compilation séparée de fonctions et de blocs fonctionnels sans références externes

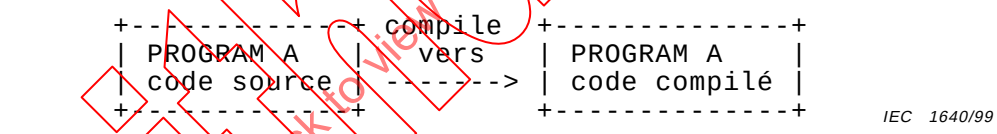


Figure 24 – Compilation d'un programme avec accès à des variables externes ou représentées directement

Ce premier niveau de compilation séparée est direct car le compilateur travaille à partir d'une source qui n'a pas de liens avec d'autres POU. Dans les cas plus complexes, une POU peut invoquer une autre fonction, ou bien un bloc fonctionnel ou un programme peuvent contenir la déclaration d'une instance d'un autre bloc fonctionnel. Ces cas exigent l'introduction du concept d'interface.

The simplest case to be considered 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 (parameters, local and external 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;
– reuse of POUs is simplified if the relocatable generated code can be associated with the source.

The separate compilation of function blocks that use external variables declared with VAR_EXTERNAL is similarly straightforward, as is the separate compilation of programs that use both VAR_EXTERNAL and directly represented variables. It is possible to compile these POUs and to resolve the unknown links upon configuration of the application.

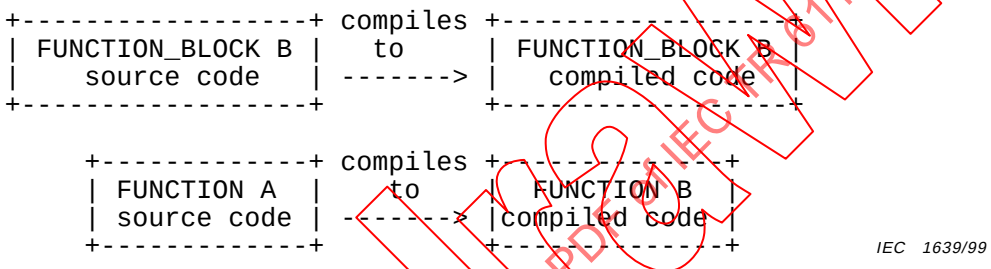


Figure 23 – Separate compilation of functions and function blocks without external reference

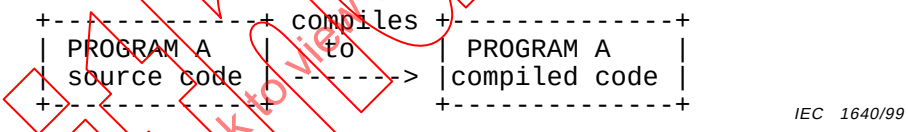


Figure 24 – Complication of a program which access to external or directly represented variables

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

5.5 Séparation entre l'interface et le corps

5.5.1 Invocation d'une fonction à partir d'une unité de programmation

Pour programmer l'invocation de la fonction B dans l'exemple de la figure 25a), il est nécessaire de savoir le nom et le type de cette fonction, et aussi la liste avec les noms et les types de ses paramètres d'entrée (VAR_INPUT). Cet ensemble d'informations constitue l'interface externe de la fonction. Pour compiler la fonction A, il suffit de connaître l'interface externe de la fonction B, comme le montre la figure 25b), même quand on ne connaît pas le code source ou le code compilé de la fonction B.

Cet exemple illustre l'importance d'une séparation nette entre le corps et l'interface d'une POU. Si, toutefois, la compilation séparée est trop difficile, l'utilisateur (qui n'est pas forcément un informaticien) peut la rejeter. Il convient donc que le PSE aide l'utilisateur autant que possible, en construisant cette interface automatiquement.

a)
```
FUNCTION A: REAL
VAR_INPUT C,D: REAL;    (* Interface externe *)
A:= B(C) + D;
END-FUNCTION
```

b)

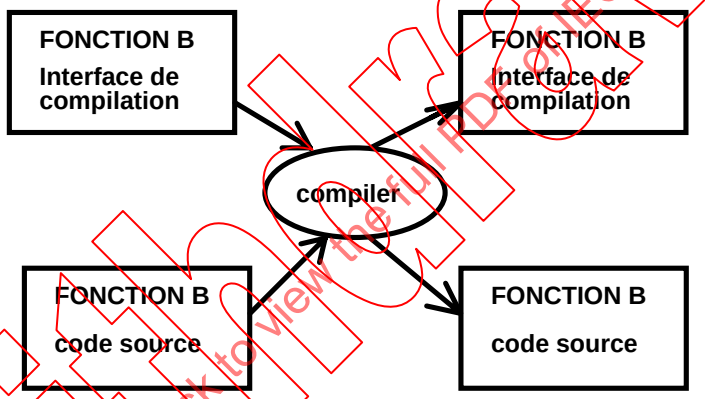


Figure 25 – Compilation d'une fonction dont le corps contient une invocation d'une autre fonction
a) Déclaration d'un exemple de fonction
b) Compilation d'un exemple de fonction

5.5.2 Déclaration et invocation d'une instance de bloc fonctionnel

Pour programmer l'invocation d'une instance de bloc fonctionnel, il est nécessaire de connaître le nom du bloc fonctionnel, les noms et les types de ses paramètres VAR_INPUT (en entrée), VAR_OUTPUT (en sortie) et VAR_IN_OUT. Cet ensemble d'informations constitue l'interface externe du bloc fonctionnel. Pour compiler, il faut déterminer la taille de la mémoire totale nécessaire pour chaque instance. On peut la calculer à partir de l'interface externe du bloc fonctionnel, et de ses déclarations de variables locales (VAR) et externes (VAR_EXTERNAL). Cet ensemble d'informations constitue l'interface de compilation. Dans la discussion ci-après, le terme «interface» se réfère à l'interface de compilation.

Pour compiler le programme F illustré dans la figure 26a), on doit connaître les interfaces des blocs fonctionnels FB1 et FB2 afin d'allouer la mémoire pour chacune de leurs instances déclarées dans le programme F. Dans cet exemple, il y a deux instances de FB1 et une instance de FB2. Cependant, comme le montre la figure 26b), les corps de ces blocs fonctionnels ne sont pas nécessaires pour compiler le programme F.