NORME INTERNATIONALE

INTERNATIONAL STANDARD

CEI
IEC
61131-3

Première édition
First edition
1993-03

Automates programmables –

Partie 3:
Langages de programmation

Programmable controllers –

Part 3:
Programming languages

Numéro de référence
Reference number
CEI/IEC 61131-3: 1993

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: *Symboles 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.

NORME INTERNATIONALE

INTERNATIONAL STANDARD

CEI
IEC
61131-3

Première édition
First edition
1993-03

Automates programmables –

Partie 3: Langages de programmation

Programmable controllers –

Part 3: Programming languages

Bureau central de la Commission Electrotechnique Internationale 3, rue de Varembé Genève Suisse

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

SOMMAIRE

CONTENTS

Articles Pages

Clause Page

Articles Pages

Tableaux

Clause Page

Tables

Articles Pages

Figures

Clause Page

Figures

Articles Pages

Clause Page

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

AUTOMATES PROGRAMMABLES –

Partie 3: Langages de programmation

AVANT-PROPOS

1) Les décisions ou accords officiels de la CEI en ce qui concerne les questions techniques, préparés par des Comités d'Etudes où sont représentés tous les Comités nationaux s'intéressant à ces questions, expriment dans la plus grande mesure possible un accord international sur les sujets examinés.

2) Ces décisions constituent des recommandations internationales et sont agréées comme telles par les Comités nationaux.

3) Dans le but d'encourager l'unification internationale, la CEI exprime le voeu que tous les Comités nationaux adoptent dans leurs règles nationales le texte de la recommandation de la CEI, dans la mesure où les conditions nationales le permettent. Toute divergence entre la recommandation de la CEI et la règle nationale correspondante doit, dans la mesure du possible, être indiquée en termes clairs dans cette dernière.

La présente partie de la Norme internationale CEI 1131 a été établie par le sous-comité 65B: Dispositifs, du comité d'études n° 65 de la CEI: Mesure et commande dans les processus industriels.

Le texte de cette partie est issu des documents suivants:

DIS	Rapport de vote
65B(BC)85	65B(BC)87

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

La CEI 1131 comprendra les parties suivantes, dont celle-ci est la troisième, présentées sous le titre général: Automates programmables.

Partie 1: 1992, Informations générales.

Partie 2: 1992, Spécifications et essais des équipements.

Partie 3: 1993, Langages de programmation.

Partie 4, Recommandations à l'utilisateur *(à l'étude).*

Partie 5, Spécification service de messagerie *(à l'étude).*

Les annexes A, B, C, D et E font partie intégrante de cette partie.
Les annexes F, G et H sont données uniquement à titre d'information.

Un rapporte technique (TR) de type 2 est en préparation. Ce TR constituera un guide de «pré-normalisation» pour la mise en oeuvre et l'application des langages de programmation définis dans la présente partie de la CEI 1131, y compris des développements tels que l'interaction système/programme et les prescriptions pour les conditions d'environnement du support de programmation.

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

FOREWORD

1) The formal decisions or agreements of the IEC on technical matters, prepared by technical committees on which all the National Committees having a special interest therein are represented, express, as nearly as possible, an international consensus of opinion on the subjects dealt with.

2) They have the form of recommendations for international use and they are accepted by the National Committees in that sense.

3) In order to promote international unification, the IEC expresses the wish that all National Committees should adopt the text of the IEC recommendation for their national rules in so far as national conditions will permit. Any divergence between the IEC recommendation and the corresponding national rules should, as far as possible, be clearly indicated in the latter.

This part of International Standard IEC 1131 has been prepared by sub-committee 65B: Devices, of IEC technical committee 65: Industrial-process measurement and control.

The text of this standard is based on the following documents:

DIS	Report on Voting
65B(CO)65	65B(CO)87

Full information on the voting for the approval of this standard can be found in the Voting Report indicated in the above table.

IEC 1131 will consist of the following parts, of which this is the third under the general title: Programmable controllers.

Part 1: 1992, General information.

Part 2: 1992, Equipment requirements and tests.

Part 3: 1993, Programming languages.

Part 4, User guidelines *(under consideration).*

Part 5, Messaging service specification *(under consideration).*

Annexes A, B, C, D and E form an integral part of this part of IEC 1131.
Annexes F, G and H are for information only.

A type 2 technical report (TR) will provide "pre-standardization" guidance for the implementation and application of the programming language defined in this part of IEC 1131, including such issues as operating system/program interaction and requirements for programming support environments.

AUTOMATES PROGRAMMABLES –

Partie 3: Langages de programmation

1 Généralités

1.1 *Domaine d'application*

Cette partie de la CEI 1131 s'applique à la représentation imprimée et affichée, à l'aide des caractères du jeu de caractères ISO/IEC 646 des langages de programmation devant être utilisés pour des automates programmables tels que définis dans la partie 1 de la CEI 1131. La représentation graphique et semi-graphique des éléments de langage définis dans cette partie est admise, mais n'est pas définie dans cette partie.

Les fonctions de programme entrée, essai, contrôle, système d'exploitation, etc., sont spécifiées dans la partie 1 de la CEI 1131.

1.2 *Références normatives*

Les documents normatifs suivants contiennent des dispositions qui, par suite de la référence qui y est faite, constituent des dispositions valables pour la présente partie de la CEI 1131. Au moment de la publication, les éditions indiquées étaient en vigueur. Tout document normatif est sujet à révision et les parties prenantes aux accords fondés sur la présente partie de la CEI 1131 sont invitées à rechercher la possibilité d'appliquer les éditions les plus récentes des documents normatifs indiqués ci-après. Les membres de la CEI et de l'ISO possèdent le registre des Normes internationales en vigueur.

CEI 50: *Vocabulaire Electrotechnique International (VEI)*

CEI 559: 1989, *Arithmétique binaire en virgule flottante pour systèmes à microprocesseurs*

CEI 617-12: 1991, *Symboles graphiques pour schémas – Partie 12: Opérateurs logiques binaires*

CEI 617-13: 1978, *Symboles graphiques pour schémas – Partie 13: Operateurs analogiques*

CEI 848: 1988, *Etablissement des diagrammes fonctionnels pour systèmes de commande*

ISO/AFNOR: 1989, *Dictionnaire de l'informatique, ISBN 2-12-486911-6*

ISO/CEI 646: 1991, *Technologie de l'information – Jeu ISO de caractères codés à 7 éléments pour l'échange d'informations (publié actuellement en anglais seulement)*

ISO 8601: 1988, *Eléments de données et formats d'échange – Echange d'information – Représentation de la date et de l'heure*

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 General

1.1 *Scope*

This part of IEC 1131 applies to the printed and displayed representation, using characters of the ISO/IEC 646 character set, of the programming languages to be used for Programmable Controllers as defined in Part 1 of IEC 1131. Graphic and semigraphic representation of the language elements which are defined in this part is allowed, but is not defined in this part.

The functions of program entry, testing, monitoring, operating system, etc., are specified in Part 1 of IEC 1131.

1.2 *Normative references*

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of IEC 1131. At the time of publication, the editions indicated were valid. All normative documents are subject to revision, and parties to agreements based on this part of IEC 1131 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

IEC 50: *International Electrotechnical Vocabulary (IEV)*

IEC 559: 1989, *Binary floating-point arithmetic for microprocessor systems*

IEC 617-12: 1991, *Graphical symbols for diagrams – Part 12: Binary logic elements*

IEC 617-13: 1978, *Graphical symbols for diagrams – Part 13: Analogue elements*

IEC 848: 1988, *Preparation of function charts for control systems*

ISO/AFNOR: 1989, *Dictionary of computer science, ISBN 2-12-486911-6*

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

ISO 8601: 1988, *Data elements and interchange formats – Information interchange – Representations of dates and times*

ISO 7185: 1990, *Technologies de l'information – Langages de programmation – Pascal (publié actuellement en anglais seulement)*

ISO 7498: 1984, *Systèmes de traitement de l'information – Interconnexion des systèmes ouverts – Modèle de référence de base*

1.3 *Définitions*

Pour les besoins de la présente partie de la CEI 1131, les définitions suivantes s'appliquent. Les définitions applicables à toutes les parties de la CEI 1131 sont données dans la partie 1.

NOTES

1 Les termes définis dans ce paragraphe sont en italiques lorsqu'ils figurent dans les corps des définitions.

2 Les définitions suivies de l'indication «(ISO)» sont tirées du *Dictionnaire de l'informatique* ISO/AFNOR.

3 En ce qui concerne les termes qui n'ont pas été définis dans la présente norme, il convient de consulter le *Dictionnaire de l'informatique* ISO/AFNOR et le *Vocabulaire Électrotechnique International.*

1.3.1 **temps absolu:** La combinaison des informations fournies par l'heure du jour et la date.

1.3.2 **chemin d'accès:** L'association d'un nom symbolique avec une variable pour les besoins d'une communication ouverte.

1.3.3 **action:** Une variable booléenne, ou ensemble d'opérations à effectuer, en même temps qu'une structure de commande associée, telle que spécifiée en 2.6.4.

1.3.4 **bloc d'action:** Un élément de langage graphique qui utilise une variable booléenne d'entrée pour déterminer la valeur d'une variable boléenne de sortie ou la condition de validation relative à une *action*, conformément à une structure de commande prédéterminée, telle que définie en 4.1.3.

1.3.5 **agrégat:** Un ensemble structuré d'objets relatifs aux données défini comme un seul *type de données*. (ISO)

1.3.6 **argument:** Synonyme de *paramètre d'entrée* ou de *paramètre de sortie.*

1.3.7 **tableau:** Un *agrégat* dont chaque constituant possède des attributs identiques, et peut être désigné sans ambiguïté par indiçage.

1.3.8 **affectation:** Un procédé permettant d'attribuer une valeur à une variable ou à un *agrégat*. (ISO)

1.3.9 **nombre basé:** Un nombre représenté dans une base spécifiée autre que la base dix.

1.3.10 **bloc fonctionnel bistable:** Un *bloc fonctionnel* disposant de deux états stables, commandé par une ou plusieurs entrées.

1.3.11 **cordon de bits:** Un élément de données composé d'un ou de plusieurs bits.

ISO 7185: 1990, *Information technology – Programming languages – Pascal*

ISO 7498: 1984, *Information processing systems – Open systems interconnection – Basic reference model*

1.3 *Definitions*

For the purposes of this part of IEC 1131, the following definitions apply. Definitions applying to all parts of IEC 1131 are given in part 1.

NOTES

1 Terms defined in this subclause are italicized where they appear in the bodies of definitions.

2 The notation "(ISO)" following a definition indicates that the definition is taken from the ISO/AFNOR *Dictionary of computer science.*

3 The ISO/AFNOR *Dictionary of computer science* and the *International Electrotechnical Vocabulary* should be consulted for terms not defined in this standard.

1.3.1 **absolute time:** The combination of time of day and date information.

1.3.2 **access path:** The association of a symbolic name with a variable for the purpose of open communication.

1.3.3 **action:** A Boolean variable, or a collection of operations to be performed, together with an associated control structure, as specified in 2.6.4.

1.3.4 **action block:** A graphical language element which utilizes a Boolean input variable to determine the value of a Boolean output variable or the enabling condition for an *action*, according to a predetermined control structure as defined in 2.6.4.5.

1.3.5 **aggregate:** A structured collection of data objects, forming a *data type.* (ISO)

1.3.6 **argument:** Synonymous with *input parameter* or *output parameter.*

1.3.7 **array:** An *aggregate* that consists of data objects, with identical attributes, each of which may be uniquely referenced by *subscripting.* (ISO)

1.3.8 **assignment:** A mechanism to give a value to a variable or to an *aggregate.* (ISO)

1.3.9 **based number:** A number represented in a specified base other than ten.

1.3.10 **bistable function block:** *A function block* with two stable states controlled by one or more inputs.

1.3.11 **bit string:** A data element consisting of one or more bits.

1.3.12 **corps:** Cette partie d'une *unité d'organisation de programme* qui spécifie les opérations à effectuer sur *les opérandes* déclarés de l'unité d'organisation de programme, lorsque son exécution est *lancée.*

1.3.13 **appel:** Un élément de langage permettant de déclencher l'exécution d'une *fonction* ou d'un *bloc fonctionnel.*

1.3.14 **cordon de caractères:** Un *agrégat* composé d'une suite ordonnée de caractères.

1.3.15 **commentaire:** Un élément de langage, permettant d'insérer dans un programme des textes quelconques sans incidence sur l'exécution de ce programme. (ISO)

1.3.16 **compilation:** Processus consistant à traduire la spécification d'une *unité d'organisation de programme* ou d'un *type de donnée* en son équivalent en langage machine ou en une autre forme intermédiaire.

1.3.17 **configuration:** Un élément de langage correspondant à un *système d'automate programmable* tel que défini dans la CEI 1131-1.

1.3.18 **bloc fonctionnel compteur:** Un *bloc fonctionnel* qui accumule une valeur pour le nombre de changements détectés sur une ou plusieurs *entrées* spécifiées.

1.3.19 **type de données:** Un ensemble de valeurs associées à l'ensemble des opérations permises sur ces valeurs. (ISO)

1.3.20 **date et heure:** La date de l'année en cours et l'heure du jour, dont la représentation est effectuée conformément aux règles fixées par ISO 8601.

1.3.21 **déclaration:** Le mécanisme que permet d'établir la définition d'un *élément de langage.* Normalement, une déclaration nécessite le rattachement d'un identificateur à l'élément de langage et l'affectation d'attributs, tels que des *types de données* et des algorithmes, à l'objet de langage concerné.

1.3.22 **délimiteur:** Un caractère ou une combinaison de caractères servant à séparer les *éléments de langage* du programme.

1.3.23 **représentation directe:** Un moyen de représenter une variable dans un système d'automate programmable à partir de laquelle une correspondance définie par le fabricant peut être déterminée directement avec un emplacement logique ou physique.

1.3.24 **double mot:** Un élément de donnée composé de 32 bits.

1.3.25 **évaluation:** Le processus qui consiste à déterminer une valeur pour une expression ou pour une *fonction*, ou pour les *sorties* d'un réseau ou d'un *bloc fonctionnel,* au cours de l'exécution d'un programme.

1.3.26 **élément de commande d'exécution:** Un *élément de langage* qui commande le flux de l'*exécution d'un programme.*

1.3.12 **body:** That portion of *a program organization unit* which specifies the operations to be performed on the declared *operands* of the program organization unit when its execution is *invoked.*

1.3.13 **call:** A language construct for *invoking* the execution of a *function* or *function block.*

1.3.14 **character string:** An *aggregate* that consists of an ordered sequence of characters.

1.3.15 **comment:** A language construct for the inclusion of text in a program and having no impact on the execution of the program. (ISO)

1.3.16 **compile:** To translate a *program organization unit* or *data type* specification into its machine language equivalent or an intermediate form.

1.3.17 **configuration:** A language element corresponding to a *programmable controller system* as defined in IEC 1131-1.

1.3.18 **counter function block:** A *function block* which accumulates a value for the number of changes sensed at one or more specified *inputs.*

1.3.19 **data type:** A set of values together with a set of permitted operations. (ISO)

1.3.20 **date and time:** The date within the year and the time of day, represented according to ISO 8601.

1.3.21 **declaration:** The mechanism for establishing the definition of a *language element.* A declaration normally involves attaching an identifier to the language element, and allocating attributes such as *data types* and algorithms to it.

1.3.22 **delimiter:** A character or combination of characters used to separate program *language elements.*

1.3.23 **direct representation:** A means of representing a variable in a programmable controller program from which a manufacturer-specified correspondence to a physical or *logical location* may be determined directly.

1.3.24 **double word:** A data element containing 32 bits.

1.3.25 **evaluation:** The process of establishing a value for an expression or a *function,* or for the *outputs* of a network or *function block*, during program execution.

1.3.26 **execution control element:** A *language element* which controls the flow of program execution.

1.3.27 **front descendant:** Le passage d'une variable booléenne de la valeur 1 à la valeur 0.

1.3.28 **fonction:** Une *unité d'organisation de programme* qui, lorsqu'elle est exécutée, fournit exactement un élément de donnée (qui peut avoir plusieurs valeurs, par exemple un *tableau* ou une *structure*), et dont l'*appel* peut servir, dans les langages littéraux, comme *opérande* dans une expression.

1.3.29 **instance de bloc fonctionnel (bloc fonctionnel):** Une *instance* d'un *type de bloc fonctionnel.*

1.3.30 **type de bloc fonctionnel:** Un *élément de langage* de programmation d'automate programmable qui consiste en: (i) la définition d'une structure de données répartie entre une entrée, une sortie et des variables internes; et (ii) un ensemble d'opérations à effectuer sur les éléments de la structure de données lorsqu'une *instance* du type de bloc fonctionnel est *appelée*.

1.3.31 **schéma en blocs fonctionnels:** Un ou plusieurs réseaux de *fonctions,* de *blocs fonctionnels,* d'éléments de données, d'*étiquettes* et d'éléments de branchement, représentés sous forme graphique.

1.3.32 **type de donnée générique:** Un *type de donnée* représentant plusieurs types de données, comme cela est spécifié en 2.3.2.

1.3.33 **champ global:** Champ d'application d'une déclaration qui a une incidence sur toutes les unités d'organisation de programmes, dans une *ressource* ou dans une *configuration.*

1.3.34 **variable globale:** Une variable dont le *champ* est *global.*

1.3.35 **adressage hiérarchique:** La *représentation directe* d'un élément de donnée comme faisant partie d'une hiérarchie physique ou logique, par exemple un point à l'intérieur d'un module situé dans un panier, lui-même logé dans une armoire, etc.

1.3.36 **identificateur:** Une combinaison de lettres, de chiffres et de caractères de soulignement tels que spécifiés en 2.1.2, qui doit nécessairement commencer par une lettre ou par un caractère de soulignement, et qui désigne un *élément de langage.*

1.3.37 **valeur initiale:** La valeur affectée à une variable, au moment du démarrage du système.

1.3.38 **paramètre d'entrée (entrée):** Un paramètre qui sert à fournir un argument à une *unité d'organisation de programme.*

1.3.39 **instance:** Une copie individuelle désignée de la structure de donnée, associée à un *type de bloc fonctionnel* ou à un *type de programme,* et qui persiste entre un *déclenchement* des opérations associées et le déclenchement suivant.

1.3.40 **nom d'instance:** Un *identificateur* associé à une instance spécifique.

1.3.41 **instanciation:** La création d'une *instance.*

1.3.27 **falling edge:** The change from 1 to 0 of a Boolean variable.

1.3.28 **function:** A *program organization unit* which, when executed, yields exactly one data element (which may be multi-valued, e.g., an *array* or *structure*), and whose *invocation* can be used in textual languages as an *operand* in an expression.

1.3.29 **function block instance (function block):** An *instance* of a *function block type.*

1.3.30 **function block type:** A programmable controller programming *language element* consisting of: (i) the definition of a data structure partitioned into input, output, and internal variables; and (ii) a set of operations to be performed upon the elements of the data structure when an *instance* of the function block type is *invoked.*

1.3.31 **function block diagram:** One or more networks of graphically represented *functions*, *function blocks*, data elements, *labels*, and connective elements.

1.3.32 **generic data type:** A *data type* which represents more than one type of data, as specified in 2.3.2.

1.3.33 **global scope:** Scope of a declaration applying to all program organization units within a *resource* or *configuration.*

1.3.34 **global variable:** A variable whose *scope* is *global.*

1.3.35 **hierarchical addressing:** The *direct representation* of a data element as a member of a physical or logical hierarchy, e.g., a point within a module which is contained in a rack, which in turn is contained in a cubicle, etc.

1.3.36 **identifier:** A combination of letters, numbers, and underline characters, as specified in 2.1.2, which begins with a letter or underline and which names a *language element.*

1.3.37 **initial value:** The value assigned to a variable at system start-up.

1.3.38 **input parameter (input):** A parameter which is used to supply an argument to a *program organization unit.*

1.3.39 **instance:** An individual, named copy of the data structure associated with a *function block type* or *program type*, which persists from one *invocation* of the associated operations to the next.

1.3.40 **instance name:** An *identifier* associated with a specific *instance.*

1.3.41 **instantiation:** The creation of an *instance.*

1.3.42 **libellé entier:** Un *libellé* qui représente directement une valeur de type SINT, INT, DINT, LINT, BOOL, BYTE, WORD, DWORD, ou LWORD, telle que définie en 2.3.1.

1.3.43 **déclenchement:** Le processus de lancement de l'exécution des opérations spécifiées dans une *unité d'organisation de programme.*

1.3.44 **mot clé:** Une unité lexicale qui caractérise un *élément de langage,* par exemple "IF".

1.3.45 **étiquette:** Une construction de langage désignant une instruction, un réseau, ou un groupe de réseaux, et comprenant un identificateur.

1.3.46 **élément de langage:** Tout élément identifié par un symbole placé à gauche d'une règle de production dans la spécification formelle donnée à l'annexe B de la présente partie de la CEI 1131.

1.3.47 **libellé:** Une unité lexicale qui représente explicitement une valeur. (ISO)

1.3.48 **champ local:** Le *champ* d'application d'une *déclaration* ou d'une *étiquette* qui ne s'applique qu'à l'*unité d'organisation de programme* dans laquelle la déclaration ou l'étiquette apparaît.

1.3.49 **emplacement logique:** L'emplacement d'une variable *adressée de manière hiérarchisée* dans un schéma qui peut ou non intégrer une relation à la structure physique des entrées, des sorties et de la mémoire de automates programmables.

1.3.50 **réel long:** Un nombre réel représenté dans un *mot long.*

1.3.51 **mot long:** Un élément de donnée composé de 64 bits.

1.3.52 **mémoire (stockage des données utilisateur):** Une unité fonctionnelle dans laquelle le programme utilisateur peut mémoriser des données et d'où il peut récupérer les données mémorisées.

1.3.53 **élément nommé:** Un élément d'une *structure,* qui est désigné par l'*identificateur* qui lui est associé.

1.3.54 **bloc fonctionnel temporisateur de déclenchement/enclechement:** Un *bloc fonctionnel* qui retarde, d'une durée spécifiée, le *front descendant (montant)* d'une *entrée* booléenne.

1.3.55 **opérande:** Un *élément de langage* sur lequel une opération est effectuée.

1.3.56 **opérateur:** Un symbole représentant l'action à exécuter dans une opération.

1.3.57 **paramètre de sortie (sortie):** Un *paramètre* servant à renvoyer le(s) résultat(s) de l'*évaluation* d'une *unité d'organisation de programme.*

1.3.58 **surchargée:** Se dit d'une opération ou d'une *fonction* capable d'intervenir sur des données de types différents, comme spécifié en 2.5.1.4.

1.3.42 **integer literal:** A *literal* which directly represents a value of type SINT, INT, DINT, LINT, BOOL, BYTE, WORD, DWORD, or LWORD, as defined in 2.3.1.

1.3.43 **invocation:** The process of initiating the execution of the operations specified in a *program organization unit.*

1.3.44 **keyword:** A lexical unit that characterizes a *language element*, e.g., "IF".

1.3.45 **label:** A language construction naming an instruction, network, or group of networks, and including an *identifier.*

1.3.46 **language element:** Any item identified by a symbol on the left-hand side of a production rule in the formal specification given in annex B of this part of IEC 1131.

1.3.47 **literal:** A lexical unit that directly represents a value. (ISO)

1.3.48 **local scope:** The *scope* of a *declaration* or *label* applying only to the *program organization unit* in which the declaration or label appears.

1.3.49 **logical location:** The location of a *hierarchically addressed* variable in a schema which may or may not bear any relation to the physical structure of the programmable controller's inputs, outputs, and memory.

1.3.50 **long real:** A real number represented in a *long word.*

1.3.51 **long word:** A 64-bit data element.

1.3.52 **memory (user data storage):** A functional unit to which the user program can store data and from which it can retrieve the stored data.

1.3.53 **named element:** An element of a *structure* which is named by its associated *identifier.*

1.3.54 **off-delay (on-delay) timer function block:** A *function block* which delays the *falling (rising) edge* of a Boolean *input* by a specified duration.

1.3.55 **operand:** A *language element* on which an operation is performed.

1.3.56 **operator:** A symbol that represents the action to be performed in an operation.

1.3.57 **output parameter (output):** A *parameter* which is used to return the result(s) of the *evaluation* of a *program organization unit.*

1.3.58 **overloaded:** With respect to an operation or *function*, capable of operating on data of different types, as specified in 2.5.1.4.

1.3.59 **sens du courant:** Le flux symbolique de l'énergie électrique dans un schéma à contacts (LD), utilisé pour indiquer la progression d'un algorithme logique de résolution.

1.3.60 **programmer:** Pour concevoir, écrire et tester des programmes utilisateur.

1.3.61 **unité d'organisation de programme:** Une *fonction*, *bloc fonctionnel*, ou programme.

NOTE - Ce terme peut se rapporter soit un *type*, soit à une *instance*.

1.3.62 **libellé réel:** Un *libellé* représentant des données de type REAL ou LREAL.

1.3.63 **ressource:** Un *élément de langage* correspondant à une "fonction de traitement de signal", ainsi qu'à son "interface homme-machine" et à ses éventuelles "fonctions d'interface de capteurs et d'actionneurs", telles que définies dans la CEI 1131-1.

1.3.64 **données non volatiles:** Des données mémorisées de telle manière que leur valeur reste invariable après une mise hors tension puis une remise sous tension.

1.3.65 **retour à la ligne:** Un élément de langage dans une *unité d'organisation de programme* qui désigne la fin d'une séquence d'exécution dans cette unité.

1.3.66 **front montant:** Le passage d'une variable booléenne de la valeur 0 à la valeur 1.

1.3.67 **champ d'application:** La partie d'un *élément de langage* à l'intérieur de laquelle s'applique une *déclaration* ou une *étiquette*.

1.3.68 **sémantique:** Les relations existant entre les éléments symboliques d'un langage de programmation et leur signification, interprétation et utilisation.

1.3.69 **représentation semigraphique:** Représentation d'informations graphiques par l'intermédiaire d'un ensemble limité de caractères.

1.3.70 **élément de donnée unitaire:** Un élément de donnée qui ne comporte qu'une seule valeur.

1.3.71 **étape:** Une situation dans laquelle le comportement d'une *unité d'organisation de programme* vis-à-vis de ses *entrées* et de ses *sorties* suit un ensemble de règles définies par les *actions* associées de l'étape.

1.3.72 **type de donnée structurée:** Un type de donnée *agrégat* qui a été déclaré à l'aide d'une déclaration de STRUCT ou de FUNCTION_BLOCK.

1.3.73 **indiciation:** Un mecanisme permettant de référencer un élément de *tableau* à l'aide d'une référence de tableau et d'une ou de plusieurs expressions qui, lorsqu'elles sont évaluées, indiquent la position de cet élément.

1.3.74 **représentation symbolique:** L'utilisation d'*identificateurs* pour nommer des variables.

1.3.75 **tâche:** Un *élément de commande d'exécution* qui permet l'exécution périodique, ou déclenchée, d'un group d'*unités d'organisations de programmes* associées.

1.3.59 **power flow:** The symbolic flow of electrical power in a ladder diagram, used to denote the progression of a logic solving algorithm.

1.3.60 **program (verb):** To design, write, and test user programs.

1.3.61 **program organization unit:** A *function, function block*, or program.

NOTE - This term may refer to either a *type* or an *instance.*

1.3.62 **real literal:** A *literal* representing data of type REAL or LREAL.

1.3.63 **resource:** A *language element* corresponding to a "signal processing function" and its "man-machine interface" and "sensor and actuator interface functions", if any, as defined in IEC 1131-1.

1.3.64 **retentive data:** Data stored in such a way that its value remains unchanged after a power down / power up sequence.

1.3.65 **return:** A language construction within a *program organization unit* designating an end to the execution sequences in the unit.

1.3.66 **rising edge:** The change from 0 to 1 of a Boolean variable.

1.3.67 **scope:** That portion of a *language element* within which a *declaration* or *label* applies.

1.3.68 **semantics:** The relationships between the symbolic elements of a programming language and their meanings, interpretation and use.

1.3.69 **semigraphic representation:** Representation of graphic information by the use of a limited set of characters.

1.3.70 **single data element:** A data element consisting of a single value.

1.3.71 **step:** A situation in which the behavior of a *program organization unit* with respect to its *inputs* and *outputs* follows a set of rules defined by the associated *actions* of the step.

1.3.72 **structured data type:** An *aggregate* data type which has been declared using a STRUCT or FUNCTION_BLOCK declaration.

1.3.73 **subscripting:** A mechanism for referencing an *array* element by means of an array reference and one or more expressions that, when evaluated, denote the position of the element.

1.3.74 **symbolic representation:** The use of *identifiers* to name variables.

1.3.75 **task:** An *execution control element* providing for periodic or triggered execution of a group of associated *program organization units.*

1.3.76 **libellé de datation:** Un *libellé* représentant des données de type TIME, DATE, TIME_OF_DAY, ou DATE_AND_TIME.

1.3.77 **transition:** La condition par laquelle une commande passe d'une ou de plusieurs *étapes* antérieures à une ou plusieurs *étapes* ultérieures le long d'une liaison dirigée.

1.3.78 **entier non signé:** Un *libellé entier* ne comportant, ni signe plus (+), ni signe moins (–).

1.3.79 **OR câblé:** Une construction matérielle permettant de réaliser la fonction booléenne OR dans le langange à contacts (LD), qui s'effectue en reliant les extrémités droites de liaison horizontales à des liaisons verticales.

1.4 *Résumé et prescriptions générales*

La présente partie de la CEI 1131 spécifie la syntaxe et la sémantique d'une série unifiée de langages de programmation pour automates programmables (AP). Cette série se compose de deux langages littéraux: le langage IL (liste d'instructions) et le langage ST (littéral structuré), et de deux langages graphiques: le langage LD (langage à contacts) et le langage FBD (diagramme fonctionnel).

Cette partie définit des éléments de schémas "diagramme fonctionnel en séquence" (SFC) destinés à structurer l'organisation interne des *programmes* d'automates programmables et de *blocs fonctionnels*. En outre, elle définit des éléments de configuration destinés à aider à l'installation de *programmes* d'automates programmables dans des systèmes d'automates programmables.

En outre, cette partie définit des caractéristiques destinées à faciliter les échanges d'informations entre des automates programmables et d'autres composants de systèmes automatisés.

Il est possible d'utiliser les éléments de langages de programmation définis dans cette partie dans un environnement de programmation interactive. La spécification de tels environnements n'est pas couverte par le domaine d'application de cette partie; cependant, un tel environnement doit être capable de produire une documentation relative à des programmes littéraux ou graphiques, dans les formats spécifiés dans cette partie.

Dans cette partie, les renseignements sont présentés "de bas en haut", c'est-à-dire que les éléments de langage les plus simples sont présentés en premier, afin de minimiser les renvois dans le texte. La suite du présent paragraphe donne un aperçu général des renseignements présentés dans cette partie et comporte quelques prescriptions générales.

1.4.1 *Modèle logiciel*

La figure 1 illustre les éléments fondamentaux d'un langage évolué, ainsi que leurs relations réciproques. Ceux-ci se composent d'éléments qui ont été *programmés* à l'aide des langages définis dans la présente partie, c'est-à-dire des *programmes* et des *blocs fonctionnels;* et d'*éléments de configuration*, à savoir: des *configurations*, des *ressources*, des *tâches*, des *variables globales* et des *chemins d'accès*, qui sont destinés à aider à l'installation de *programmes* d'automates programmables dans des systèmes d'automates programmables.

1.3.76 **time literal:** A *literal* representing data of type TIME, DATE, TIME_OF_DAY, or DATE_AND_TIME.

1.3.77 **transition:** The condition whereby control passes from one or more predecessor *steps* to one or more successor steps along a directed link.

1.3.78 **unsigned integer:** An *integer literal* not containing a leading plus (+) or minus (–) sign.

1.3.79 **wired OR:** A construction for achieving the Boolean OR function in the LD language by connecting together the right ends of horizontal connectives with vertical connectives.

1.4 *Overview and general requirements*

This part of IEC 1131 specifies the syntax and semantics of a unified suite of programming languages for programmable controllers (PCs). These consist of two textual languages, IL (Instruction List) and ST (Structured Text), and two graphical languages, LD (Ladder Diagram) and FBD (Function Block Diagram).

Sequential Function Chart (SFC) elements are defined for structuring the internal organization of programmable controller *programs* and *function blocks*. Also, *configuration elements* are defined which support the installation of programmable controller *programs* into programmable controller systems.

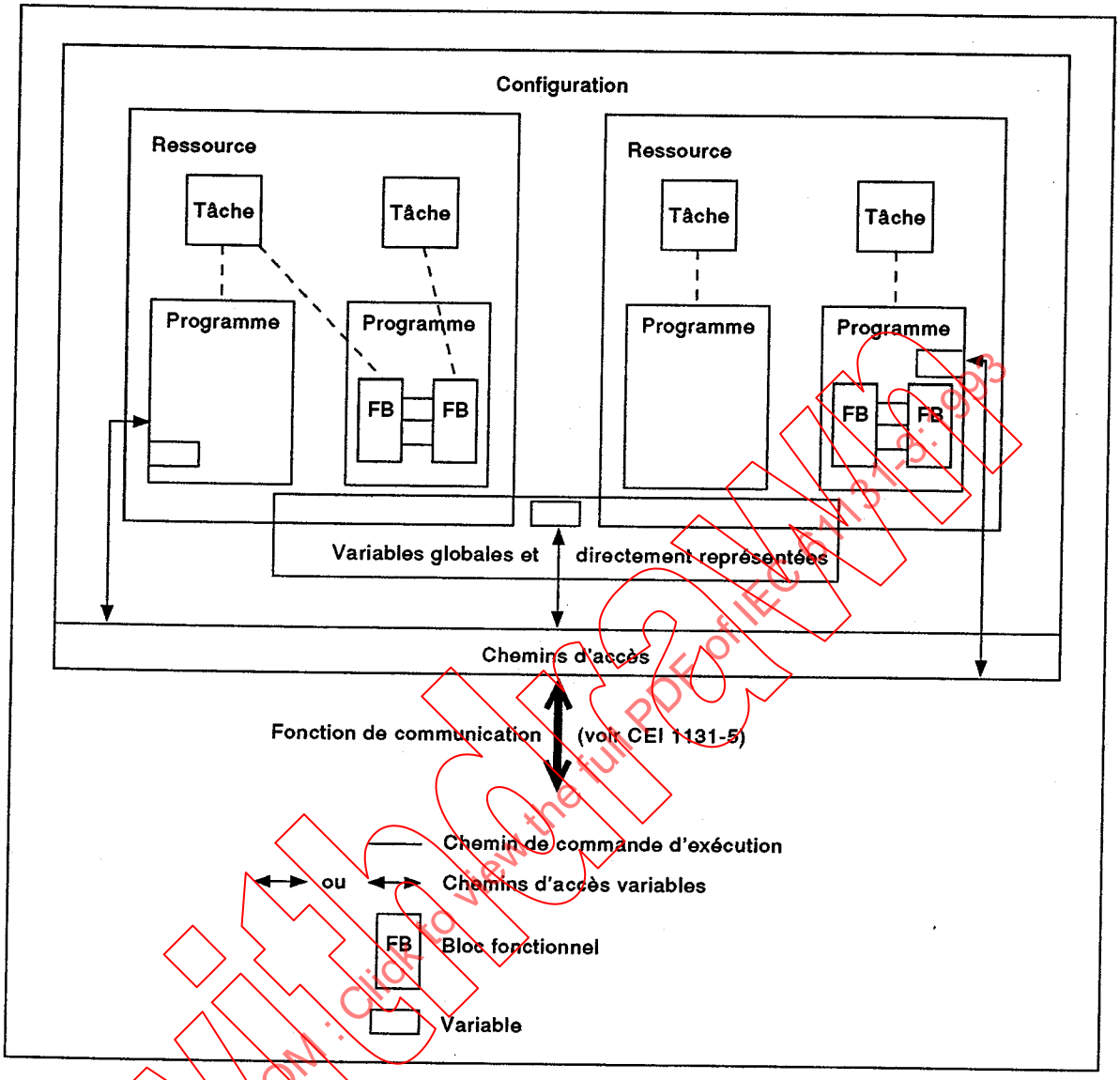
In addition, features are defined which facilitate communication among programmable controllers and other components of automated systems.

The programming language elements defined in this part may be used in an interactive programming environment. The specification of such environments is beyond the scope of this part; however, such an environment shall be capable of producing textual or graphic program documentation in the formats specified in this part.

The material in this part is arranged in "bottom-up" fashion, that is, simpler language elements are presented first, in order to minimize forward references in the text. The remainder of this subclause provides an overview of the material presented in this part and incorporates some general requirements.

1.4.1 *Software model*

The basic high-level language elements and their interrelationships are illustrated in figure 1. These consist of elements which are *programmed* using the languages defined in this part, that is, *programs* and *function blocks*; and *configuration elements*, namely, *configurations*, *resources*, *tasks*, *global variables*, and *access paths*, which support the installation of programmable controller *programs* into programmable controller systems.



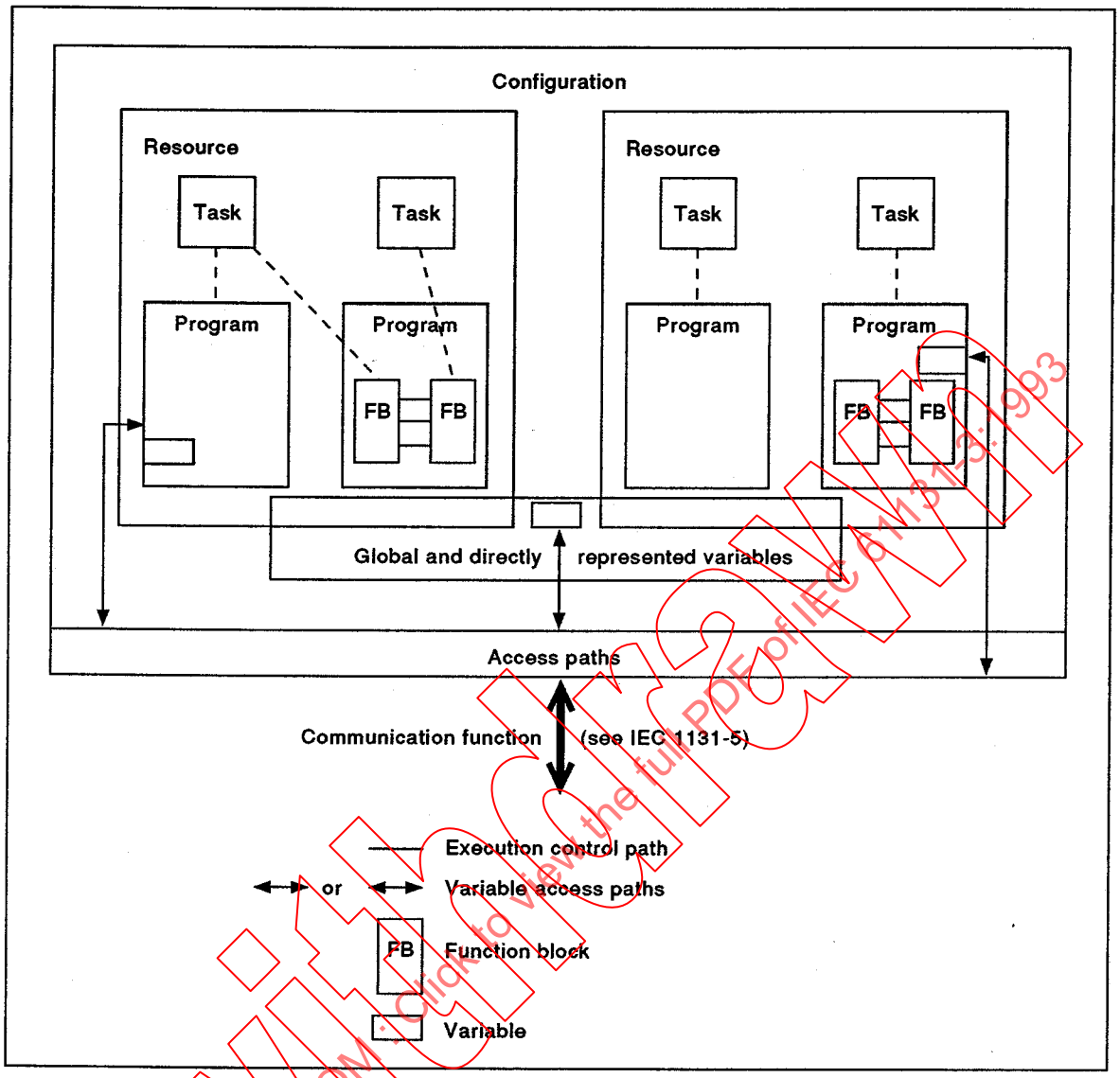
NOTES

1 Cette figure n'est donnée qu'à titre indicatif. La représentation graphique n'est pas normative.

2 Dans une configuration à ressource unique, il n'est pas nécessaire de représenter explicitement la ressource.

Figure 1 – Modèle logiciel

Une *configuration* se définit comme l'élément de langage qui correspond à un *système d'automate programmable* tel que défini dans la CEI 1131-1. Une *ressource* correspond à une "fonction de traitement du signal" ainsi qu'à ses éventuelles fonctions "interface homme-machine" et "interface capteurs et actionneurs", telles que définies dans la CEI 1131-1. Une configuration contient une ou plusieurs *ressources*, chacune de ces ressources contenant un ou plusieurs *programmes* exécutés sous le contrôle de zéro ou plusieurs *tâches*. Un programme peut ne contenir aucun *bloc fonctionnel*, ou il peut contenir un ou plusieurs blocs fonctionnels ou autres éléments de programme tels que définis dans la présente partie.



NOTES

1 This figure is illustrative only. The graphical representation is not normative.

2 In a configuration with a single resource, the resource need not be explicitly represented.

Figure 1 – Software model

A *configuration* is the language element which corresponds to a *programmable controller system* as defined in IEC 1131-1. A *resource* corresponds to a "signal processing function" and its "man-machine interface" and "sensor and actuator interface" functions (if any) as defined in IEC 1131-1. A *configuration* contains one or more *resources*, each of which contains one or more *programs* executed under the control of zero or more *tasks*. A *program* may contain zero or more *function blocks* or other language elements as defined in this part.

Les *configurations* et les *ressources* peuvent être lancées et arrêtées par l'intermédiaire des fonctions "interface opérateur", "programmation, essai et contrôle", ou "système d'exploitation", définies dans la CEI 1131-1. Le lancement d'une *configuration* doit entraîner l'initialisation de ses *variables globales,* conformément aux règles spécifiées en 2.4.2, suivie du lancement de toutes les autres *ressources* présentes dans la configuration. Le lancement d'une ressource doit entraîner l'initialisation de toutes les *variables* dans la ressource, suivie de la validation de toutes les *tâches* dans la ressource. L'arrêt d'une *ressource* doit entraîner l'invalidation de toutes ses *tâches,* alors que l'arrêt d'une *configuration* doit entraîner l'arrêt de toutes ses *ressources.* Les mécanismes de commande de *tâches* sont définis en 2.7.2, alors que les mécanismes relatifs au démarrage et à l'arrêt des *configurations* et des *ressources,* par l'intermédiaire de fonctions de communication, sont définis dans la CEI 1131-5.

Les *programmes,* les *ressources,* les *variables globales,* les *chemins d'accès* (ainsi que leurs autorisations d'accès correspondantes) et les *configurations* peuvent être chargés et supprimés par la "fonction communication" définie dans la CEI 1131-1. Le chargement ou la suppression d'une *configuration* ou d'une *ressource* doivent être équivalents au chargement ou à la suppression de tous les éléments qu'elle contient.

Les *chemins d'accès* et leurs autorisations d'accès correspondantes sont définis en 2.7.1.

La projection des éléments de langage, définis dans le présent paragraphe, sur les objets de communication, est définie dans la CEI 1131-5.

1.4.2 *Modèle de communication*

La figure 2 illustre les différentes manières dont les valeurs de variables peuvent être échangées entre les éléments logiciels.

Comme l'illustre la figure 2a, des valeurs de variables, dans un programme, peuvent être directement échangées en reliant la sortie d'un élément de programme à l'entrée d'un autre élément de programme. Cette liaison est indiquée de façon explicite dans les langages graphiques et de façon implicite dans les langages littéraux.

Des valeurs de variables peuvent être échangées entre programmes dans la même configuration par l'intermédiaire de *variables globales,* comme, par exemple, la variable *x* illustrée à la figure 2b. Ces variables doivent être déclarées comme GLOBAL dans la configuration, et comme EXTERNAL dans les programmes, comme spécifié en 2.4.3.

Comme l'illustre la figure 2c, les valeurs de variables peuvent être échangées entre les différentes parties d'un programme dans la même configuration, ou dans des configurations différentes, ou entre un programme d'automate programmable et un autre programme, à l'aide des blocs de fonction de communication définis dans la CEI 1131-5 et décrits en 2.5.2.3.5. En outre, les systèmes d'automates programmables ou autres peuvent transférer des données qui sont rendues disponibles par des *chemins d'accès,* telles qu'illustrées à la figure 2d, à l'aide des mécanismes définis dans la CEI 1131-5.

Configurations and *resources* can be started and stopped via the "operator interface", "programming, testing, and monitoring", or "operating system" functions defined in IEC 1131-1. The starting of a *configuration* shall cause the initialization of its *global variables* according to the rules given in 2.4.2, followed by the starting of all the resources in the configuration. The starting of a resource shall cause the initialization of all the *variables* in the resource, followed by the enabling of all the *tasks* in the resource. The stopping of a *resource* shall cause the disabling of all its *tasks*, while the stopping of a configuration shall cause the stopping of all its resources. Mechanisms for the control of tasks are defined in 2.7.2, while mechanisms for the starting and stopping of *configurations* and *resources* via communication functions are defined in IEC 1131-5.

Programs, resources, global variables, access paths (and their corresponding access privileges), and *configurations* can be loaded or deleted by the "communication function" defined in IEC 1131-1. The loading or deletion of a *configuration* or *resource* shall be equivalent to the loading or deletion of all the elements it contains.

Access paths and their corresponding access privileges are defined in 2.7.1.

The mapping of the language elements defined in this subclause on to communication objects is defined in IEC 1131-5.

1.4.2 *Communication model*

Figure 2 illustrates the ways that values of variables can be communicated among software elements.

As shown in figure 2a, variable values within a program can be communicated directly by connection of the output of one program element to the input of another. This connection is shown explicitly in graphical languages and implicitly in textual languages.

Variable values can be communicated between programs in the same configuration via *global* variables such as the variable *x* illustrated in figure 2b. These variables shall be declared as GLOBAL in the configuration, and as EXTERNAL in the programs, as specified in 2.4.3.

As illustrated in figure 2c, the values of variables can be communicated between different parts of a program, between programs in the same or different configurations, or between a programmable controller program and a non-programmable controller system, using the communication function blocks defined in IEC 1131-5 and described in 2.5.2.3.5. In addition, programmable controllers or non-programmable controller systems can transfer data which is made available by *access paths*, as illustrated in figure 2d, using the mechanisms defined in IEC 1131-5.

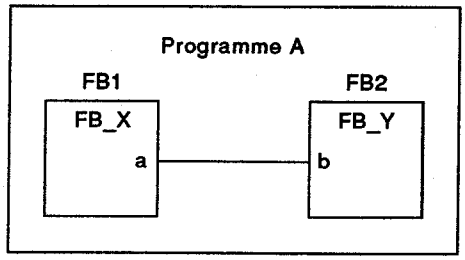


Figure 2a – Liaison de flux de données dans un programme

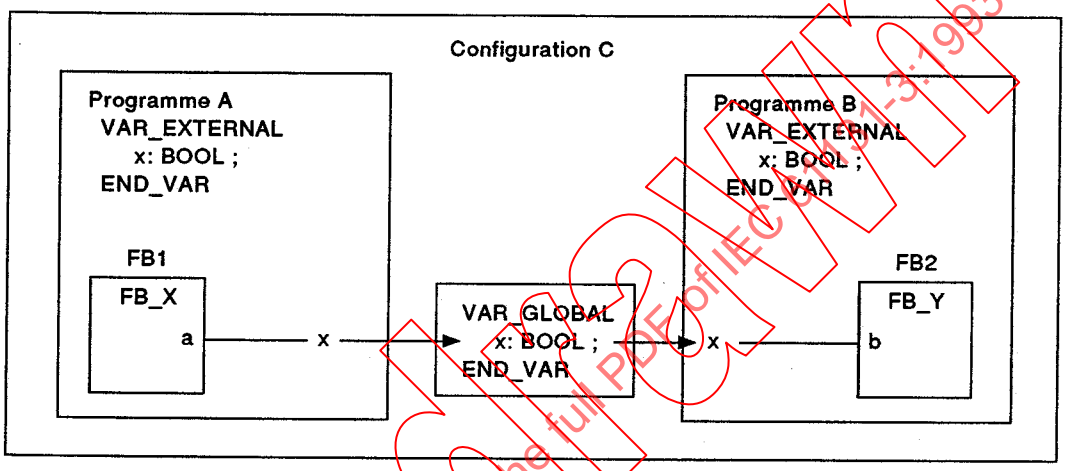


Figure 2b – Communication par l'intermédiaire de variables globales

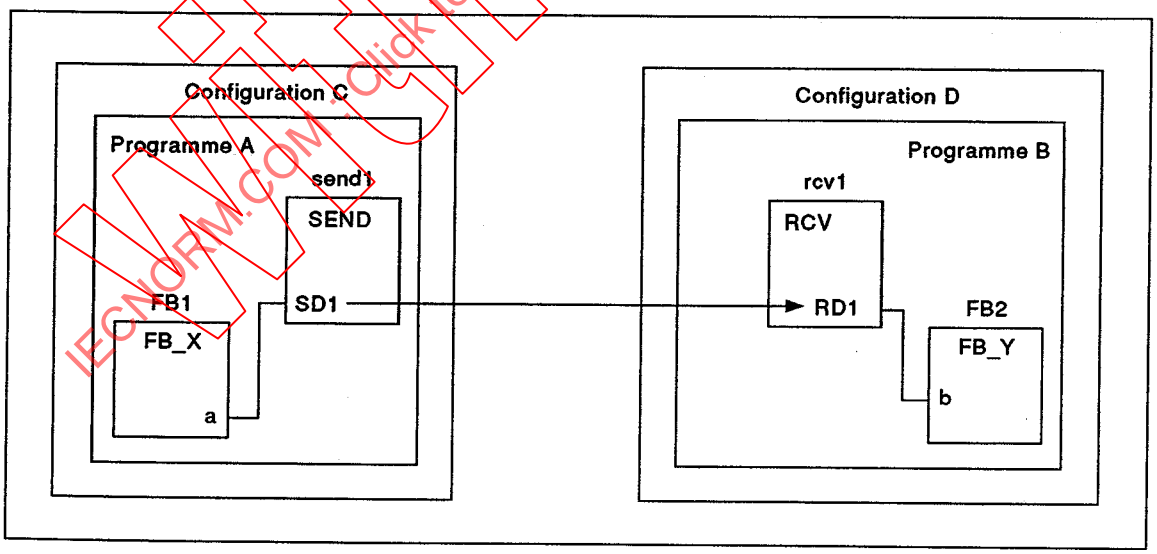


Figure 2c – Blocs fonctionnels de communication

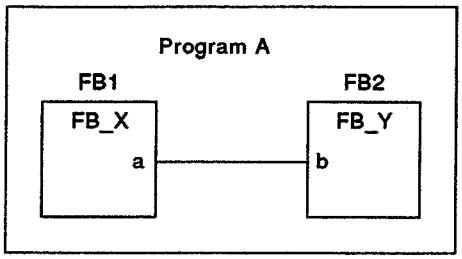


Figure 2a – Data flow connection within a program

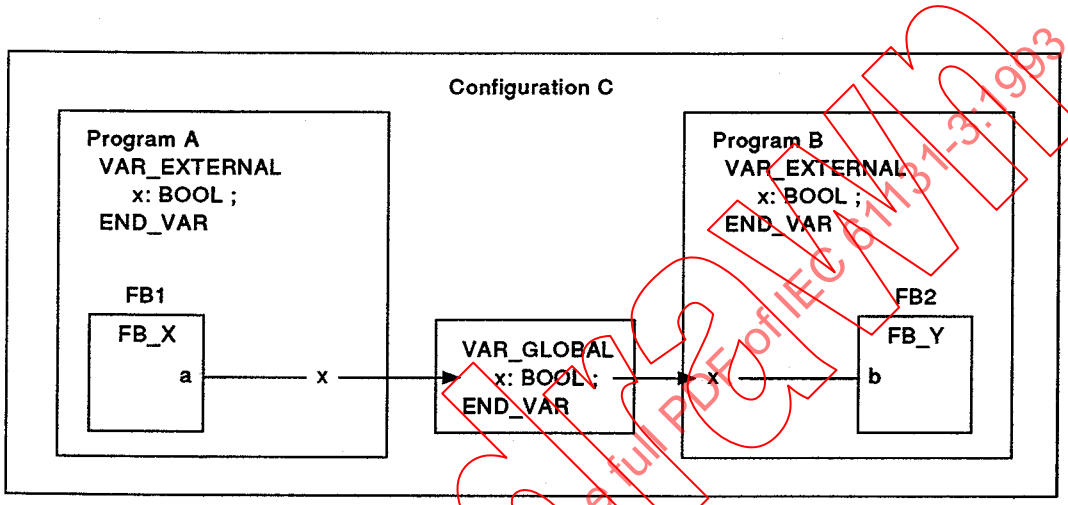


Figure 2b – Communication via GLOBAL variables

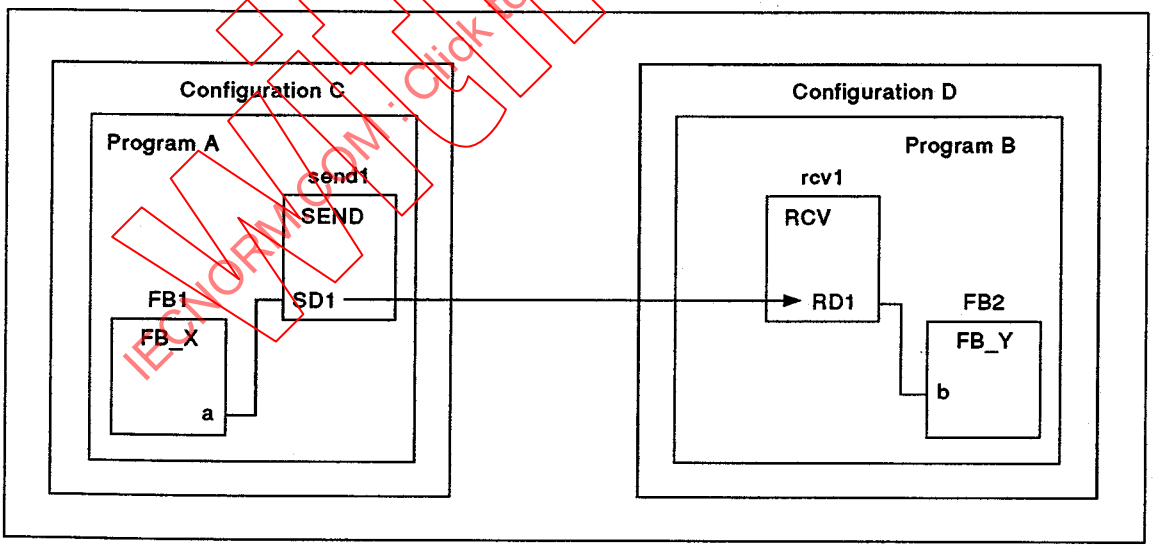


Figure 2c – Communication function blocks

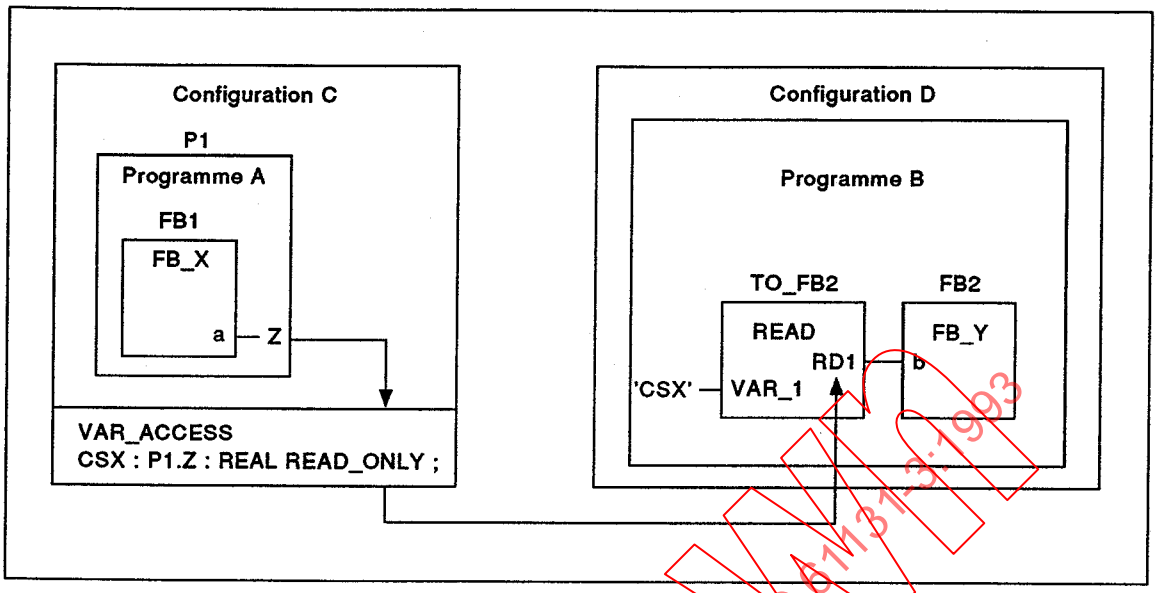


Figure 2d – Communication par l'intermédiaire de chemins d'accès

NOTES

1 Cette figure n'est fournie qu'à titre indicatif. La représentation graphique n'est pas normative.

2 Dans ces exemple, les configurations C et D sont respectivement considérées comme ayant une seule ressource.

3 Les details relatifs aux blocs fonctionnels de communication ne sont pas représentés dans cette figure. Se reporter à 2.5.2.3.5 et à la CEI 1131-5.

4 Comme spécifié en 2.7, les chemins d'accès peuvent être déclarés sur des variable directement représentées, sur des variables globales, ou sur des variables d'entrée, de sortie, ou internes programme.

5 La CEI 1131-5 spécifie les moyens dont peuvent disposer les systèmes d'automates programmables et d'autres systèmes pour l'écriture et la lecture de variables.

Figure 2 – Modèle de communication

1.4.3 *Modèle de communication*

Les éléments des langages de programmation d'automates programmables et les paragraphes au sein desquels ils figurent dans la présente partie sont classés comme suit:

- Types de données (2.3)
- Unités d'organisation de programmes (2.5)
  - Fonctions (2.5.1)
  - Blocs fonctionnels (2.5.2)
  - Programmes (2.5.3)
- Eléments de schémas "diagramme fonctionnel en séquence" (SFC) (2.6)
- Eléments de configuration (2.7)
  - Variables globales 2.7.1)
  - Ressources (2.7.1)
  - Tâches (2.7.2)
  - Chemins d'accès (2.7.1)

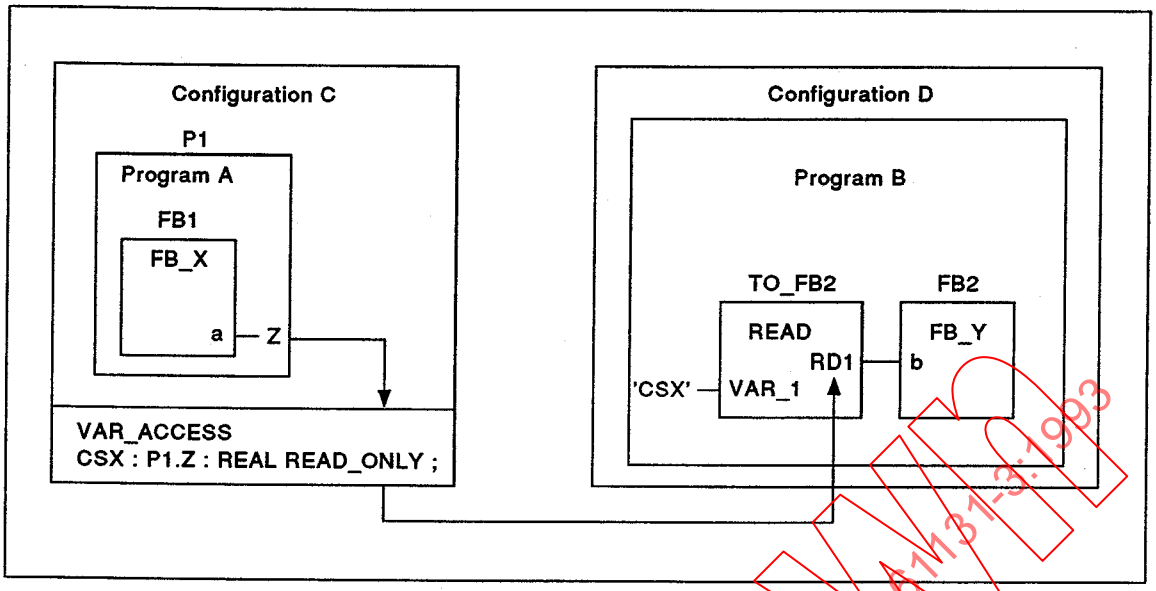


Figure 2d – Communication via access paths

NOTES

1 This figure is illustrative only. The graphical representation is not normative.

2 In these examples, configurations c and d are each considered to have a single resource.

3 The details of the communication function blocks are not shown in this figure. See 2.5.2.3.5 and IEC 1131-5.

4 As specified in 2.7, access paths can be declared on directly represented variables, global variables, or program input, output, or internal variables.

5 IEC 1131-5 specifies the means by which both PC and non-PC systems can use access paths for reading and writing of variables.

Figure 2 – Communication model

1.4.3 *Programming model*

The elements of programmable controller programming languages, and the subclauses in which they appear in this part, are classified as follows:

- Data types (2.3)
- Program organization units (2.5)
  - Functions (2.5.1)
  - Function blocks (2.5.2)
  - Programs (2.5.3)
- Sequential Function Chart (SFC) elements (2.6)
- Configuration elements (2.7)
  - Global variables (2.7.1)
  - Resources (2.7.1)
  - Tasks (2.7.2)
  - Access paths (2.7.1)

Comme l'illustre la figure 3, la combinaison de ces éléments doit obéir aux règles suivantes:

1) Les *types de données* dérivés doivent être déclarés conformément aux indications de 2.3.3, à l'aide des types de données standards, spécifiés en 2.3.1 et 2.3.2, ainsi que de tout type de donnée précédemment dérivé.

2) Les *fonctions* dérivées peuvent être déclarées conformément aux indications de 2.5.1.3, à l'aide de types de données standards ou dérivés, des fonctions standards définies au paragraphe 2.5.1.5, ainsi que de tout type de fonction précédemment dérivé. Cette déclaration doit utiliser les mécanismes définis pour le langage IL, ST, LD ou FBD.

3) Les *blocs fonctionnels* dérivés peuvent être déclarés conformement aux indications de 2.5.2.2, à l'aide de types de données et de fonctions standards ou dérivés, des blocs fonctionnels définis en 2.5.2.3, anis que de tout bloc fonctionnel précédemment dérivé. Cette déclaration doit utiliser les mécansimes définis pour la langage IL, ST ou FBD, et peut inclure des éléments de schéma "diagramme fonctionnel en séquence" (SFC), tels que définis en 2.6.

4) Un *programme* doit être déclaré conformément aux indications de 2.5.3, à l'aide de types de données, de fonctions et de blocs fonctionnels standards ou dérivés. Cette déclaration doit utiliser le mécanisme défini pour le langage IL, ST, LD ou FBD, et peut inclure des éléments de schéma "diagramme fonctionnel en séquence" (SFC), tels que définis en 2.6.

5) Les *programmes* peuvent être combinés en *configurations,* à l'aide des éléments définis en 2.7, c'est-à-dire des *variables globales,* des *ressources,* des *tâches* et des *chemins d'accès.*

La référence aux types de données, aux fonctions et aux blocs fonctionnels "précédemment dérivés" dans les règles précédentes, est destinée à garantir que, lorsqu'un tel élément a été déclaré, sa définition est disponible, par exemple, dans une "bibliothèque" d'éléments dérivés, pour une utilisation dans des dérivations ultérieures. Par conséquent, la déclaration d'un type d'élément dérivé ne doit pas être contenue dans la déclaration d'un autre type d'élément dérivé.

Il est possible d'utiliser un langage de programmation autre que ceux définis dans la présente norme dans la déclaration d'une *fonction* ou d'un *bloc fonctionnel.* Il est nécessaire de définir dans la présente norme le moyen par lequel un programme utilisateur, écrit dans l'un des langages définis dans la présente norme, déclenche l'exécution d'une telle fonction dérivée ou d'un tel bloc fonctionnel dérivé, et accède aux données associées à une telle fonction ou à un tel bloc fonctionnel.

As shown in figure 3, the combination of these elements shall obey the following rules:

1) Derived *data types* shall be declared as specified in 2.3.3, using the standard data types specified in 2.3.1 and 2.3.2 and any previously derived data types.

2) Derived *functions* can be declared as specified in 2.5.1.3, using standard or derived data types, the standard functions defined in 2.5.1.5, and any previously derived functions. This declaration shall use the mechanisms defined for the IL, ST, LD or FBD language.

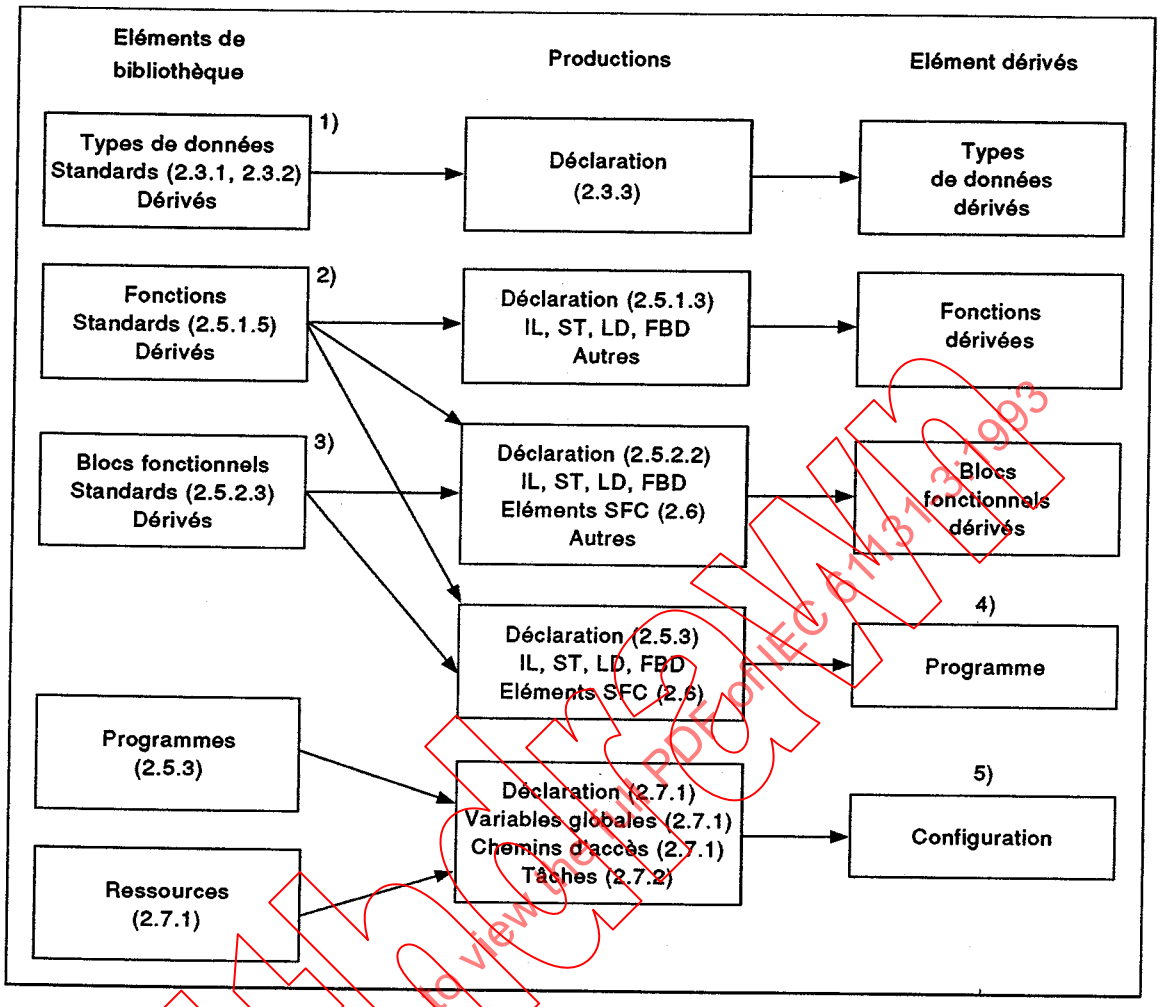
3) Derived *function blocks* can be declared as specified in 2.5.2.2, using standard or derived data types and functions, the standard function blocks defined in 2.5.2.3, and any previously derived function blocks. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements as defined in 2.6.

4) A *program* shall be declared as specified in 2.5.3, using standard or derived data types, functions, and function blocks. This declaration shall use the mechanisms defined for the IL, ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements as defined in 2.6.

5) *Programs* can be combined into *configurations* using the elements defined in 2.7, that is, *global variables, resources, tasks,* and *access paths.*

Reference to "previously derived" data types, functions, and function blocks in the above rules is intended to imply that once such a derived element has been declared, its definition is available, e.g., in a "library" of derived elements, for use in further derivations. Therefore, the declaration of a derived element type shall not be contained within the declaration of another derived element type.

A programming language other than one of those defined in this standard may be used in the declaration of a *function* or *function block.* The means by which a user program written in one of the languages defined in this standard invokes the execution of, and accesses the data associated with, such a derived function or function block shall be as defined in this standard.



NOTES

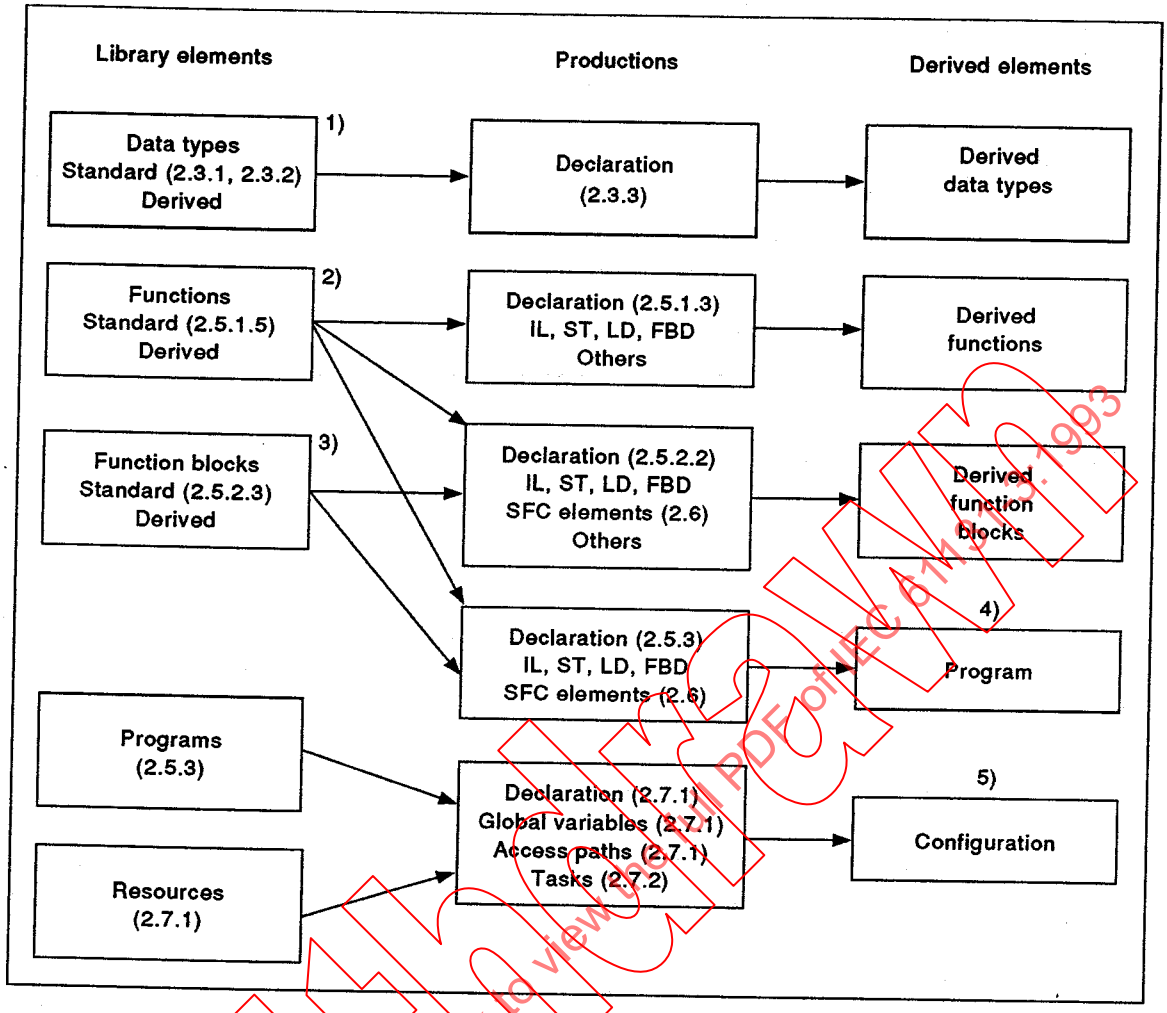
1 Les numéros 1) à 5) se rapportent aux alinéas correspondants de 1.4.3.

2 Les types de données sont utilisés dans toutes les productions. Pour plus de clarté, les liaisons correspondantes ont été omises dans cette figure.

Figure 3 – Combinaison d'éléments de langages pour automates programmables:
Langage à contacts-LD (4.2)
Schéma en blocs fonctionnels-FBD (4.3)
Liste d'instructions-IL (3.2)
Littéral structuré-ST (3.3)
Autres: Autres langages de programmation (1.2.3)

1.5 *Conformité*

Le présent paragraphe définit les prescriptions auxquelles doivent satisfaire les systèmes et les programmes d'automates programmables qui sont réputés conformes aux prescriptions de cette partie de la CEI 1131.



NOTES

1 The parenthesized numbers 1) to 5) refer to corresponding paragraphs in 1.4.3.

2 *Data types* are used in all productions. For clarity, the corresponding linkages are omitted in this figure.

Figure 3 – Combination of programmable controller language elements
LD-Ladder Diagram (4.2)
FBD-Function Block Diagram (4.3)
IL-Instruction List (3.2)
ST-Structured Text (3.3)
Others: Other programming languages (1.4.3)

1.5 *Compliance*

This subclause defines the requirements which shall be met by programmable controller systems and programs which claim compliance with this part of IEC 1131.

1.5.1 *Systèmes d'automates programmables*

Un système d'automate programmable, tel qu'il a été défini dans le CEI 1131-1, qui prétend se conformer, totalement ou partiellement, aux exigences de cette partie de la CEI 1131, ne doit être considéré comme tel que lorsque les conditions ci-dessous sont remplies.

Un énoncé de conformité doit être inclus dans la documentation accompagnant le système, ou doit être produit par le système lui-même. La formulation de l'énoncé de conformité doit être comme suit:

"Ce système est conforme aux prescriptions de la CEI 1131-3, en ce qui concerne les caractéristiques de langage suivantes:"

suivie d'un ensemble de tableaux de conformité présentant le format suivant:

Titre du tableau

Tableau n°	Caractéristique n°	Description des caractéristiques
...	...	...

Les numéros et la description des caractéristiques doivent être pris dans les tableaux donnés dans le paragraphe correspondant dans la présente partie de la CEI 1131. Les titres des tableaux doivent être pris parmi ceux indiqués dans le tableau ci-dessous.

Titre du tableau	Pour les caractéristiques indiquées en:
Eléments communs	Article 2
Eléments littéraux communs	Paragraphe 3.1
Eléments de langage IL	Paragraphe 3.2.1 à 3.2.3
Eléments de langage ST	Paragraphe 3.3.1 à 3.3.2.4
Eléments graphiques communs	Paragraphe 4.1 à 4.1.4
Eléments de langage LD	Paragraphe 4.2 à 4.2.6
Eléments de langage FBD	Paragraphe 4.3 à 4.3.3

Un système d'automate programmable, conforme aux exigences de la présente partie, eu égard à un langage défini dans la présente partie:

a) ne doit pas nécessiter l'inclusion d'éléments de langage de remplacement ou supplémentaires, pour accomplir l'une des fonctions spécifiées dans la présente partie;

b) doit être accompagné d'un document qui spécifie les valeurs de tous les paramètres propres à une application, énumérées dans l'annexe D;

c) doit être capable de déterminer si un élément de langage utilisateur est conforme ou non à toute prescription de la présente partie, lorsqu'une telle non-conformité n'est pas imputable à l'une des erreurs indiquées à l'annexe E, et de signaler cette détermination à l'utilisateur. Dans le cas où le système n'examine pas la totalité de l'unité

1.5.1 *Programmable controller systems*

A programmable controller system, as defined in IEC 1131-1, which claims to comply, wholly or partially, with the requirements of this part of IEC 1131 shall do so only as described below.

A compliance statement shall be included in the documentation accompanying the system, or shall be produced by the system itself. The form of the compliance statement shall be:

"This system complies with the requirements of IEC 1131-3, for the following language features:"

followed by a set of compliance tables in the following format:

Table title

Table No.	Feature No.	Features description
...	...	...

Table and feature numbers and descriptions are to be taken from the tables given in the relevant subclauses of this part of IEC 1131. Table titles are to be taken from the following table.

Table title	For features in:
Common elements	Clause 2
Common textual elements	Subclause 3.1
IL language elements	Subclauses 3.2.1 to 3.2.3
ST language elements	Subclauses 3.3.1 to 3.3.2.4
Common graphical elements	Subclauses 4.1 to 4.1.4
LD language elements	Subclauses 4.2 to 4.2.6
FBD language elements	Subclauses 4.3 to 4.3.3

A programmable controller system complying with the requirements of this part with respect to a language defined in this part:

a) shall not require the inclusion of substitute or additional language elements in order to accomplish any of the features specified in this part;

b) shall be accompanied by a document that specifies the values of all implementation-dependent parameters as listed in annex D;

c) shall be able to determine whether or not a user's language element violates any requirement of this part, where such a violation is not designated an error in annex E, and report the result of this determination to the user. In the case where the system does not examine the whole program organization unit, the user shall be notified that

d'organisation de programme, l'utilisateur doit être avisé du fait que la détermination est incomplète, si aucune non-conformité n'a été décelée dans la partie examinée de l'unité d'organisation de programme;

d) doit traiter chaque non-conformité utilisateur qui correspond à une erreur indiquée à l'annexe E de la présente partie, en procédant selon au moins une des façons suivantes:

1) il doit y avoir une déclaration dans un document d'accompagnement indiquant que l'erreur n'a pas été signalée;

2) le système doit signaler, au cours de la préparation du programme pour l'exécution, que cette erreur risque de se produire;

3) le système doit signaler l'erreur, au cours de la préparation du programme pour l'exécution;

4) le système doit signaler l'erreur pendant l'exécution du programme et lancer les procédures appropriées de traitement des erreurs, définies par le système ou par l'utilisateur;

et si une éventuelle non-conformité, reconnue comme une erreur, est traitée de la manière décrite au point 1) de l'alinéa d) ci-dessus, une note se rapportant à chacun de ces traitements doit apparaître dans une section séparée du document d'accompagnement;

e) doit être accompagné d'un document qui décrit séparément toutes caractéristiques prises en charge par le système, mais qui sont interdites ou non spécifiées dans la présente partie. De telles caractéristiques doivent être décrites comme des "extensions au langage <langage> tel que défini dans la CEI 1131-3";

f) doit être capable de traiter, d'une manière similaire à celle spécifiée pour les erreurs, toute utilisation de toute extension de ce type;

g) doit être capable de traiter, d'une manière similaire à celle spécifiée pour les erreurs, toute utilisation de l'une des caractéristiques propres à une application, spécifiées à l'annexe D;

h) ne doit utiliser aucun nom de type de donnée, de fonction, ou de bloc fonctionnel standard défini dans la présente partie, pour des caractéristiques définies par le fabricant et dont la fonctionnalité est différente de celle décrite dans la présente partie;

i) doit être accompagné d'un document définissant, sous la forme spécifiée à l'annexe A, la syntaxe formelle de tous les éléments de langage littéral, prise en charge par le système.

L'expression "être capable de" est utilisée dans le présent paragraphe pour permettre la mise en application d'un commutateur logiciel grâce auquel l'utilisateur peut commander la signalisation des erreurs.

Dans les cas où la compilation ou la saisie d'un programme est suspendue en raison de l'insuffisance des tableaux, etc., une détermination incomplète du type "aucune non-conformité n'a été décelée, mais l'examen est incomplet" satisfera aux prescriptions de ce paragraphe.

1.5.2 *Programmes*

Un programme d'automate programmable, conforme aux prescriptions de la CEI 1131-3:

a) ne doit utiliser que les caractéristiques spécifiées dans la présente partie pour le langage spécifique utilisé;

the determination is incomplete whenever no violations have been detected in the portion of the program organization unit examined;

d) shall treat each user violation that is designated an error in annex E in at least one of the following ways:

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

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

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

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

and if any violations that are designated as errors are treated in the manner described in d)1) above, then a note referencing each such treatment shall appear in a separate section of the accompanying document;

e) shall be accompanied by a document that separately describes any features accepted by the system that are prohibited or not specified in this part. Such features shall be described as being "extensions to the <language> language as defined in IEC 1131-3";

f) shall be able to process in a manner similar to that specified for errors any use of any such extension;

g) shall be able to process in a manner similar to that specified for errors any use of one of the implementation-dependent features specified in annex D;

h) shall not use any of the standard data type, function or function block names defined in this part for manufacturer-defined features whose functionality differs from that described in this part;

i) shall be accompanied by a document defining, in the form specified in annex A, the formal syntax of all textual language elements supported by the system.

The phrase "be able to" is used in this subclause to permit the implementation of a software switch with which the user may control the reporting of errors.

In cases where compilation or program entry is aborted due to some limitation of tables, etc., an incomplete determination of the kind "no violations were detected, but the examination is incomplete" will satisfy the requirements of this subclause.

1.5.2 *Programs*

A programmable controller program complying with the requirements of IEC 1131-3:

a) shall use only those features specified in this part for the particular language used;

b) ne doit utiliser aucune caractéristique identifiée comme une extension du langage;

c) ne doit appuyer sur aucune interprétation particulière des caractéristiques propres à une application.

Les résultats rendus par un programme conforme doivent être identiques lorsqu'ils sont traités par un système conforme qui prend en charge les caractéristiques utilisées par le programme, sauf si ces résultats sont influencés par le séquencement de l'exécution du programme, l'utilisation de caractéristiques propres à une application (telles qu'énumérées dans l'annexe D) dans le programme, et l'exécution de procédures de traitement d'erreurs.

2 Eléments communs

Le présent article définit les éléments littéraux et graphiques qui sont communs à tous les langages de programmation d'automates programmables spécifiés dans cette partie de la CEI 1131.

2.1 *Utilisation de caractères imprimés*

2.1.1 *Jeu de caractères*

Les langages littéraux et les éléments littéraux des langages graphiques doivent être représentés en fonction du "tableau des codes de base" du jeu de caractères ISO/IEC 646.

Le codage de caractères, à partir des jeux de caractères nationaux ou étendus (8 bits), doit être conforme à l'ISO/IEC 646.

Le *jeu de caractères requis,* illustré en tant que caractéristique 1 dans le tableau 1, se compose de tous les caractères des colonnes 3 à 7 du "tableau des codes de base", présenté en tant que tableau 1 dans l'ISO/IEC 646, à l'exception des caractères minuscules et des positions de caractères réservées ou disponibles en option, pour l'utilisation dans des jeux de caractères nationaux.

Le fabricant doit choisir une option (a ou b) pour chacune des caractéristiques (3a, b) à (6a, b) du tableau 1, conformément aux règles suivantes:

– Le "signe livre sterling" (£) doit être utilisé à la place du "signe numéro" (#), lorsque le premier signe occupe la position de caractère 2/3 de l'application nationale du jeu de caractères ISO/IEC 646.

– Le "signe monétaire" doit être utilisé à la place du signe "dollar" ($), lorsque le premier occupe la position de caractère 2/4 d'une application nationale du jeu de caractères ISO 646.

– Lorsque la position de caractère 7/12 dans le jeu de caractères ISO 646 est utilisée par un autre caractère dans un jeu de caractères national, "le point d'exclamation" (!) doit être utilisé en position 2/1 pour représenter des lignes verticales.

– Pour la délimitation des indices, les parenthèses à droite et à gauche "( )" doivent être utilisées à la place des crochets à droite et à gauche "[ ]" lorsque ces derniers occupent les positions de caractères d'une application nationale du jeu de caractères ISO 646.

NOTE - L'utilisation de caractères provenant de jeux de caractères nationaux constitue un exemple d'extenison type de la présente norme.

b) shall not use any features identified as extensions to the language;

c) shall not rely on any particular interpretation of implementation-dependent features.

The results produced by a complying program shall be the same when processed by any complying system which supports the features used by the program, except as these results are influenced by program execution timing, the use of implementation-dependent features (as listed in annex D) in the program, and the execution of error handling procedures.

2 Common elements

This clause defines textual and graphic elements which are common to all the programmable controller programming languages specified in this part of IEC 1131.

2.1 *Use of printed characters*

2.1.1 *Character set*

Textual languages and textual elements of graphic languages shall be represented in terms of the "Basic code table" of the ISO/IEC 646 character set.

The encoding of characters from national or extended (8-bit) character sets shall be consistent with ISO/IEC 646.

The *required character set* shown as feature 1 in table 1 consists of all the characters in columns 3 to 7 of the "Basic code table" given as table 1 in ISO/IEC 646, except for lowercase letters and those character positions which are reserved or optionally available for use in national character sets.

The manufacturer shall support one option (a or b) for each of features (3a,b) to (6a,b) of table 1, according to the following rules:

– The "pound sign" (£) shall be used in place of the "number sign" (#) when the former occupies character position 2/3 of a national implementation of the ISO/IEC 646 character set.

– The "currency sign" shall be used in place of the "dollar sign" ($) when the former occupies character position 2/4 of a national implementation of the ISO/IEC 646 character set.

– When the 7/12 character position in the ISO/IEC 646 character set is used by another character in a national set, the "exclamation mark" (!) at position 2/1 shall be used to represent vertical lines.

– For delimitation of subscripts, the left and right parentheses "( )" shall be used in place of the left and right brackets "[ ]" when the latter occupy character positions of a national implementation of the ISO/IEC 646 character set.

NOTE - The use of characters from national character sets is a typical extension of this standard.

Tableau 1 – Caractéristiques des jeux de caractères

N°	Description
1	Jeu de caractères requis (voir 2.1.1)
2	Caractères miniscules
3a 3b	Signe numéro (#) ou Signe livre sterling (£)
4a 4b	Signe dollar ($) ou Signe monétaire
5a 5b	Barre verticale (|) ou Point d'exclamation (!)
6a 6b	Délimiteur d'indice: Crochet gauche et droit "[ ]" ou Parenthèses gauche et droite "( )"
NOTE - Lorsque des lettres minuscules (caractéristique 2) sont prises en charge, les caractères majuscules représentant des mêmes lettres ne doivent pas être une signification particulière dans les éléments de langage (sauf dans les commentaires tels que définis en 2.1.5 et dans les libellés de cordons de caractères et des variables du type STRING, respectivement définis en 2.2.2 et 2.3.1); par exemple, les identificateurs "abcd", "ABCD" et "aBCd" doivent être interprétés de façon identique.	

2.1.2 *Identificateurs*

Un *identificateur* est un cordon de lettres, de chiffres et de caractères de soulignement qui doit commencer par une lettre ou par un caractère de soulignement.

Les caractères de soulignement doivent avoir une signification particulière dans les identificateurs; par exemple, "A_BCD" et "AB_CD" doivent être interprétés comme étant des identificateurs différents. Les caractères de soulignement multiples en tête ou intégrés ne sont pas autorisés.

Les identificateurs ne doivent pas comporter de caractères d'espaces intercalaires (SP).

Au moins six caractères uniques doivent être pris en charge dans tous les systèmes qui acceptent l'utilisation d'identificateurs; par exemple, "ABCDE1" doit être considéré comme différent de "ABCDE2" dans tous ces types de systèmes.

Les caractéristiques des identificateurs et des exemples se trouvent à la figure 2.

Tableau 2 – Caractéristiques des identificateurs

N°	Description des caractéristiques	Exemples
1	Majuscules et nombres	IW215 IW215Z QX75 IDENT
2	Majuscule et minuscule, nombres, caractères de soulignement intégrés	Tous les exemples ci-dessus, plus: LIM_SW_5 LimSw5 abcd ab_Cd
3	Majuscule et minuscule, nombres, caractères de soulignement en tête ou intégrés	Tous les exemples ci-dessus, plus: _MAIN _12V7

Table 1 – Character set features

No	Description
1	Required character set (see 2.1.1)
2	Lower case characters
3a 3b	Number sign (#) or Pound sign (£)
4a 4b	Dollar sign ($) or Currency sign
5a 5b	Vertical bar (|) or Exclamation mark (!)
6a 6b	Subscript delimiters: Left and right brackets "[ ]" or Left and right parentheses "( )"

NOTE - When lower-case letters (feature 2) are supported, the case of letters shall not be significant in language elements (except within terminal symbols as defined in annexes A and B, comments as defined in 2.1.5, string literals as defined in 2.2.2, and variables of type STRING as defined in 2.3.1), e.g., the identifiers "abcd", "ABCD" and "aBCd" shall be interpreted identically.

2.1.2 *Identifiers*

An *identifier* is a string of letters, digits, and underline characters which shall begin with a letter or underline character.

Underlines shall be significant in identifiers, e.g., "A_BCD" and "AB_CD" shall be interpreted as different identifiers. Multiple leading or multiple embedded underlines are not allowed.

Identifiers shall not contain imbedded space (SP) characters.

At least six characters of uniqueness shall be supported in all systems which support the use of identifiers, e.g., "ABCDE1" shall be interpreted as different from "ABCDE2" in all such systems.

Identifier features and examples are shown in table 2.

Table 2 – Identifier features

No.	Feature description	Examples
1	Upper case and numbers	IW215 IW215Z QX75 IDENT
2	Upper and lower case, numbers, embedded underlines	All the above plus: LIM_SW_5 LimSw5 abcd ab_Cd
3	Upper and lower case, numbers, leading or embedded underlines	All the above plus: _MAIN _12V7

2.1.3 *Mots clés*

Les *mots clés* sont des combinaisons uniques de caractères utilisés comme des éléments syntaxiques individuels, tels que définis à l'annexe B. Tous les mots clés utilisés dans la présente partie sont énumérés à l'annexe C. Les mots clés ne doivent pas comporter de caractères d'espaces intercalaires. Les mots clés indiqués à l'annexe C ne doivent servir à aucune autre fin, comme par exemple les noms de variables ou extensions, tels que définis en 1.5.1.

NOTE - Les organismes nationaux de normalisation peuvent publier des tableaux de traductions des mots clés donnés à l'annexe C.

2.1.4 *Utilisation des espaces*

L'utilisateur doit être autorisé à insérer un ou plusieurs espaces (position de code 2/0 dans le jeu de caractères ISO/IEC 646) n'importe où dans le texte de programmes d'automates programmables, sauf dans des mots clés, des libellés, des identificateurs, ou des combinaisons de délimiteurs (pour des commentaires tels que ceux définis ci-après).

2.1.5 *Commentaires*

Les commentaires de l'utilisateur doivent être respectivement délimités, à leur début et à leur fin, par les combinaisons de caractères spéciaux "(*" et "*)"comme indiqué au tableau 3. Sauf dans le langage IL tel que défini au paragraphe 3.2, des commentaires doivent être autorisés n'importe où dans le programme, lorsque des espaces sont autorisés, à l'exception des libellés de cordons de caractères définis au paragraphe 2.2.2. Les commentaires ne doivent avoir aucune signification syntaxique ni sémantique, dans aucun des langages définis dans la présente partie.

Les commentaires imbriqués, comme par exemple (* (* IMBRIQUE *) *), ne sont pas autorisés.

Tableau 3 – La caractéristique commentaire

N°	Description des caractéristiques	Exemples
1	Commentaires	(**********) (* Un commentaire encadré *) (**********)

2.2 *Représentation externe des données*

Les représentations externes des données, dans les divers langages de programmation d'automates programmables, doivent se composer de libellés numériques, de cordons de caractères et de libellés de datation.

2.2.1 *Libellés numériques*

Il existe deux catégories de libellés numériques: les libellés entiers et les libellés réels. Un libellé numérique se définit comme un nombre décimal ou comme un nombre basé. Le nombre maximal de chiffres, pour chaque type de libellé numérique, doit être suffisant pour exprimer l'étendue entière et la précision des valeurs de tous les types de données qui sont représentés par le libellé dans une mise en oeuvre donnée.

2.1.3 *Keywords*

Keywords are unique combinations of characters utilized as individual syntactic elements as defined in annex B. All keywords used in this part are listed in annex C. Keywords shall not contain imbedded spaces. The keywords listed in annex C shall not be used for any other purpose, e.g., variable names or extensions as defined in 1.5.1.

NOTE - National standards organizations can publish tables of translations of the keywords given in annex C.

2.1.4 *Use of spaces*

The user shall be allowed to insert one or more spaces (code position 2/0 in the ISO/IEC 646 character set) anywhere in the text of programmable controller programs except within keywords, literals, identifiers, or delimiter combinations (e.g., for comments as defined below).

2.1.5 *Comments*

User comments shall be delimited at the beginning and end by the special character combinations "(*" and "*)", respectively, as shown in table 3. Except in the IL language as defined in 3.2, comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals as defined in 2.2.2. Comments shall have no syntactic or semantic significance in any of the languages defined in this part.

Nested comments are not allowed, e.g., (* (* NESTED *) *).

Table 3 – Comment feature

No.	Feature description	Examples
1	Comments	(**************************) (* A framed comment *) (**************************)

2.2 *External representation of data*

External representations of data in the various programmable controller programming languages shall consist of numeric literals, character strings, and time literals.

2.2.1 *Numeric literals*

There are two classes of numeric literals: integer literals and real literals. A numeric literal is defined as a decimal number or a based number. The maximum number of digits for each kind of numeric literal shall be sufficient to express the entire range and precision of values of all the data types which are represented by the literal in a given implementation.

Les caractères de soulignement uniques (_), insérés entre les chiffres d'un libellé numérique, ne doivent pas être significatifs. Aucune autre utilisation de caractères de soulignement dans des libellés numériques n'est autorisée.

Les libellés décimaux doivent être représentés en notation décimale conventionnelle. Les libellés réels doivent se distinguer par la présence d'une virgule décimale. La présence d'un exposant indique la puissance entière de dix par laquelle le nombre précédent doit être multiplié pour obtenir la valeur représentée. Les libellés décimaux et leurs exposants peuvent être précédés d'un signe (+ ou –).

Les libellés entiers peuvent être également représentés en base 2, 8 ou 16. La base doit être en notation décimale. Pour la base 16, un ensemble étendu de chiffres composés des lettres A à F doit être utilisé, avec respectivement la signification conventionnelle de décimal 10 à 15. Les nombres basés ne doivent pas être précédés d'un signe (+ ou –).

Les données booléennes doivent être représentées par des libellés entiers, avec la valeur zéro (0) ou un (1) ou les mots clés FAUX et VRAI respectivement.

Des caractéristiques de libellés numériques et des exemples sont donnés au tableau 4.

Tableau 4 – Libellés numériques

N°	Description des caractéristiques	Exemples
1	Libellés entiers	–12 0 123_456 +986
2	Libellés réels	–12.0 0.0 0.4560 3.14159_26
3	Libellés réels avec des des exposants	–1.34E-12 ou –1.34e-12 1.0E+6 ou 1.0e+6 1.234E6 ou 1.234e6
4	Libellés en base 2	2#1111_1111 (255 décimal) 2#1110_0000 (240 décimal)
5	Libellés en base 8	8#377 (255 décimal) 8#340 (240 décimal)
6	Libellés en base 16	16#FF ou 16#ff (255 décimal) 16#E0 ou 16#e0 (240 décimal)
7	Zéro et un booléens	0 1
8	FAUX et VRAI booléens	FAUX VRAI
NOTE - Les mots clés FAUX et VRAI correspondent respectivement aux valeurs booléennes de 0 et 1.		

2.2.2 *Libellés de cordons de caractères*

Un libellé de cordon de caractères est une séquence de zéro à plusieurs caractères, précédée et terminée par une apostrophe ('). Dans les cordons de caractères, la combinaison à trois caractères du signe dollar ($), suivie de deux chiffres hexadécimaux, doit être interprétée comme la représentation hexadécimale du code de caractères à huit bits conformément au tableau 5. En outre, des combinaisons à deux caractères, commençant par le signe dollar, doivent être interprétées conformément au tableau 6, lorsqu'elles apparaissent dans des cordons de caractères.

Single underline characters (_) inserted between the digits of a numeric literal shall not be significant. No other use of underline characters in numeric literals is allowed.

Decimal literals shall be represented in conventional decimal notation. Real literals shall be distinguished by the presence of a decimal point. An exponent indicates the integer power of ten by which the preceding number is to be multiplied to obtain the value represented. Decimal literals and their exponents can contain a preceding sign (+ or -).

Integer literals can also be represented in base 2, 8, or 16. The base shall be in decimal notation. For base 16, an extended set of digits consisting of the letters A through F shall be used, with the conventional significance of decimal 10 through 15, respectively. Based numbers shall not contain a leading sign (+ or -).

Boolean data shall be represented by integer literals with the value zero (0) or one (1), or the keywords FALSE or TRUE, respectively.

Numeric literal features and examples are shown in table 4.

Table 4 – Numeric literals

No.	Feature description	Examples
1	Integer literals	–12 0 123_456 +986
2	Real literals	–12.0 0.0 0.4560 3.14159_26
3	Real literals with exponents	–1.34E-12 or –1.34e-12 1.0E+6 or 1.0e+6 1.234E6 or 1.234e6
4	Base 2 literals	2#1111_1111 (255 decimal) 2#1110_0000 (240 decimal)
5	Base 8 literals	8#377 (255 decimal) 8#340 (240 decimal)
6	Base 16 literals	16#FF or 16#ff (255 decimal) 16#E0 or 16#e0 (240 decimal)
7	Boolean zero and one	0 1
8	Boolean FALSE and TRUE	FALSE TRUE

NOTE – The keywords FALSE and TRUE correspond to Boolean values of 0 and 1, respectively.

2.2.2 *Character string literals*

A character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character ('). In character strings, the three-character combination of the dollar sign ($) followed by two hexadecimal digits shall be interpreted as the hexadecimal representation of the eight-bit character code, as shown in table 5. Additionally, two-character combinations beginning with the dollar sign shall be interpreted as shown in table 6 when they occur in character strings.

Tableau 5 – Caractéristique des libellés de cordons de caractères

N°	Exemple	Explication
1	''	Cordon vide (longueur zéro)
	'A'	Cordon de longueur un comportant le caractère unique A
	' '	Cordon de longueur un comportant le caractère "espace"
	'$''	Cordon de longueur un comportant le caractère "apostrophe"
	'RL' '$0D$0A'	Cordons de longueur deux comportant les caractères CR et LF
	'$$1.00'	Cordon de longueur cinq qui devrait être imprimé en tant que "$1.00"

Tableau 6 – Combinaisons à deux chiffres dans les cordons de caractères

N°	Combinaison	Interprétation lors de l'impression
2	$$	Signe dollar
3	$'	Apostrophe
4	$L ou $l	Changement de ligne
5	$N ou $n	Changement de ligne
6	$P ou $p	Changement de page (page)
7	$R ou Sr	Retour du curseur
8	$T ou $t	Tabulation
NOTE - Le caractère "changement de ligne" fournit un moyen, indépendant de la mise en oeuvre, pour définir la fin d'une ligne de données relatives à une E/S physique et de fichier; en ce qui concerne l'impression, cela a pour effet la fin d'une ligne de données et la reprise de l'impression au début de la ligne suivante.		

2.2.3 *Libellés de datation*

La nécessité de fournir des représentations externes relatives à deux types distincts de données temporelles est reconnue; les données relatives à la *durée*, pour la mesure et le contrôle du temps écoulé pour un événement de commande donné, et les données relatives à *l'heure du jour* (qui peuvent également inclure des informations relatives à la date), pour la synchronisation du début et de la fin d'un événement de commande par rapport à une référence de temps absolu.

Les libellés de durée et d'heure du jour doivent être délimités à gauche par les mots clés définis en 2.2.3.1 et 2.2.3.2.

2.2.3.1 *Durée*

Les données relatives à la durée doivent être délimitées à gauche par le mot clé T#, TIME#, t#, ou time#. La représentation des données relatives à la durée, en termes de jours, heures, minutes, secondes et millisecondes, ou toute combinaison de ces derniers, doit être prise en charge conformément au tableau 7. L'unité de temps la moins significative peut être écrite en notation réelle sans exposant.

Table 5 – Character string literal feature

No.	Example	Explanation
1	''	Empty string (length zero)
	'A'	String of length one containing the single character A
	' '	String of length one containing the "space" character
	'$''	String of length one containing the "single quote" character
	'RL' '$0D$0A'	Strings of length two containing CR and LF characters
	'$$1.00'	String of length five which would print as "$1.00"

Table 6 – Two-character combinations in character strings

No.	Combination	Interpretation when printed
2	$$	Dollar sign
3	$'	Single quote
4	$L or $l	Line feed
5	$N or $n	Newline
6	$P or $p	Form feed (page)
7	$R or Sr	Carriage return
8	$T or $t	Tab
NOTE - The "newline" character provides an implementation-independent means of defining the end of a line of data for both physical and file I/O; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line.		

2.2.3 *Time literals*

The need to provide external representations for two distinct types of time-related data is recognized: *duration* data for measuring or controlling the elapsed time of a control event, and *time of day* data (which may also include date information) for synchronizing the beginning or end of a control event to an absolute time reference.

Duration and time of day literals shall be delimited on the left by the keywords defined in 2.2.3.1 and 2.2.3.2.

2.2.3.1 *Duration*

Duration data shall be delimited on the left by the keyword T#, TIME#, t#, or time#. The representation of duration data in terms of days, hours, minutes, seconds, and milli-seconds, or any combination thereof, shall be supported as shown in table 7. The least significant time unit can be written in real notation without exponent.

Les unités des libellés de temps peuvent être séparées par des caractères de soulignement.

Le "dépassement" de l'unité la plus significative d'un libellé de temps est permis; par exemple, la notation T#25h_15m est permise.

Les unités de temps, comme par exemple secondes, millisecondes, etc. peuvent être représentées en majuscules et en minuscules.

Tableau 7 – Caractéristique de libellés de temps

N°	Description de la caractéristique	Exemples
1a	Libellés de temps sans caractères de soulignement: Préfixe court	T#14ms T#14.7s T#14.7m T#14.7h t#14.7d t#25h15m t#5d14h12m18s3.5ms
1b	Préfixe long	TIME#14ms time#14.7s
2a	Libellés de temps avec caractères de soulignement: Préfixe court	t#25h_15m t#5d_14h_12m_18s_3.5ms
2b	Préfixe long	TIME#25h_15m time#5d_14h_12m_18s_3.5ms

2.2.3.2 *Heure du jour et date*

Les mots clés préfixes, relatifs aux libellés heure du jour et date doivent être conformes au tableau 8. La représentation des informations relatives à l'heure du jour et à la date, telle qu'elle est représentée à la figure 9, doit être conforme aux spécifications de la norme ISO 8601.

Tableau 8 – Libellés de date et heure du jour

N°	Description des caractéristiques	Mot de préfixe
1	Libellés de date (préfixe long)	DATE#
2	Libellés de date (préfixe court)	D#
3	Libellés d'heure du jour (préfixe long)	TIME_OF_DAY#
4	Libellés d'heure du jour (préfixe court)	TOD#
5	Libellés de date et d'heure du jour (préfixe long)	DATE_AND_TIME#
6	Libellés de date et d'heure du jour (préfixe court)	DT#

The units of duration literals can be separated by underline characters.

"Overflow" of the most significant unit of a duration literal is permitted, e.g., the notation T#25h_15m is permitted.

Time units, e.g., seconds, milliseconds, etc., can be represented in upper- or lower- case letters.

Table 7 – Duration literal features

No.	Feature description	Examples
1a	Duration literals without underlines: Short prefix	T#14ms T#14.7s T#14.7m T#14.7h t#14.7d t#25h15m t#5d14h12m18s3.5ms
1b	Long prefix	TIME#14ms time#14.7s
2a	Duration literals with underlines: Short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms
2b	Long prefix	TIME#25h_15m time#5d_14h_12m_18s_3.5ms

2.2.3.2 *Time of day and date*

Prefix keywords for time of day and date literals shall be as shown in table 8. As illustrated in table 9, representation of time-of-day and date information shall be as specified in ISO 8601.

Table 8 – Date and time of day literals

No.	Feature description	Prefix keyword
1	Date literals (long prefix)	DATE#
2	Date literals (short prefix)	D#
3	Time of day literals (long prefix)	TIME_OF_DAY#
4	Time of day literals (short prefix)	TOD#
5	Date and time literals (long prefix)	DATE_AND_TIME#
6	Date and time literals (short prefix)	DT#

Tableau 9 – Exemples de libellés de date et heure du jour

Notation de préfixes longs	Notation de préfixes courts
DATE#1984-06-25 date#1984-06-25	D#1984-06-25 d#1984-06-25
TIME_OF_DAY#15:36:55.36 time_of_day#15:36:55.36	TOD#15:36:55.36 tod#15:36:55.36
DATE_AND_TIME#1984-06-25-15:36:55.36 date_and_time#1984-06-25-15:36:55.36	DT#1984-06-25-15:36:55.36 dt#1984-06-25-15:36:55.36

2.3 *Types de données*

Un certain nombre de types de données élémentaires (prédéfinies) sont reconnus par la présente norme. En outre, des types de données génériques sont définis pour l'utilisation dans la définition de fonctions surchargées (voir 2.5.1.4). Un mécanisme permettant à l'utilisateur ou au fabricant de spécifier des types de données supplémentaires est également défini.

2.3.1 *Types de données élémentaires*

Les types de données élémentaires, le mot clé relatif à chaque type de donnée, le nombre de bits par élément d'information, et la plage de valeurs relative à chaque type de donnée élémentaire doivent être tels que définis au tableau 10.

Table 9 – Examples of date and time of day literals

Long prefix notation	Short prefix notation
DATE#1984-06-25 date#1984-06-25	D#1984-06-25 d#1984-06-25
TIME_OF_DAY#15:36:55.36 time_of_day#15:36:55.36	TOD#15:36:55.36 tod#15:36:55.36
DATE_AND_TIME#1984-06-25-15:36:55.36 date_and_time#1984-06-25-15:36:55.36	DT#1984-06-25-15:36:55.36 dt#1984-06-25-15:36:55.36

2.3 *Data types*

A number of elementary (pre-defined) data types are recognized by this standard. Additionally, generic data types are defined for use in the definition of overloaded functions (see 2.5.1.4). A mechanism for the user or manufacturer to specify additional data types is also defined.

2.3.1 *Elementary data types*

The elementary data types, keyword for each data type, number of bits per data element, and range of values for each elementary data type shall be as shown in table 10.

Tableau 10 – Types de données élémentaires

N°	Mot clé	Type de donnée	Bits	Étendue
1	BOOL	Booléen	1	Note 8
2	SINT	Entier court	8	Note 2
3	INT	Entier	16	Note 2
4	DINT	Entier double	32	Note 2
5	LINT	Entier long	64	Note 2
6	USINT	Entier court non signé	8	Note 3
7	UINT	Entier non signé	16	Note 3
8	UDINT	Entier double non signé	32	Note 3
9	ULINT	Entier long non signé	64	Note 3
10	REAL	Nombre réels	32	Note 4
11	LREAL	Réels longs	64	Note 5
12	TIME	Durée	Note 1	Note 6
13	DATE	Date (uniquement)	Note 1	Note 6
14	TIME_OF_DAY ou TOD	Heure du jour (uniquement)	Note 1	Note 6
15	DATE_AND_TIME ou DT	Date et heure du jour	Note 1	Note 6
16	STRING	Cordon de caractères de longueur variable	Note 1	Note 7
17	BYTE	Cordon de bits de longueur 8	8	Note 7
18	WORD	Cordon de caractères de longueur 16	16	Note 7
19	DWORD	Cordon de caractères de longueur 32	32	Note 7
20	LWORD	Cordon de caractères de longueur 64	64	Note 7

NOTES

1 La longueur de ces données dépend de l'application concernée.

2 L'étendue des valeurs, relatives à des variables de ce type de donnée, est comprise entre -(2**(Bits-1)) et (2**(Bits-1))-1.

3 L'étendue des valeurs, relatives à des variables de ce type de donnée, est comprise entre 0 et (2**Bits)-1.

4 L'étendue des valeurs, relatives à des variables de ce type de donnée doit être telle que définie dans la CEI 559 pour le format de virgule flottante à largeur binaire unique.

5 L'étendue des valeurs, relatives à des variables de ce type de donnée, doit être telle que définie dans la CEI 559 pour le format de virgule flottante à double largeur binaire.

6 L'étendue des valeurs relatives à des variables de ce type de donnée dépend de l'application concernée.

7 Une étendue numérique de valeurs ne s'applique pas à ce type de donnée.

8 Les valeurs que peut prendre variables de ce type de donnée doivent être 0 et 1, correspondant respectivement aux mots clés FAUX et VRAI.

2.3.2 *Types de données génériques*

Outre les types de données indiqués au tableau 10, la hiérarchie des types de données génériques indiqués au tableau 11 doit être utilisée conformément aux prescriptions de 2.5.1.4, dans la spécification des entrées et des sorties surchargées des fonctions et des blocs fonctionnels standards. Les types de données génériques sont identifiés par le préfixe "ANY".

Table 10 – Elementary data types

No.	Keyword	Data type	Bits	Range
1	BOOL	Boolean	1	Note 8
2	SINT	Short integer	8	Note 2
3	INT	Integer	16	Note 2
4	DINT	Double integer	32	Note 2
5	LINT	Long integer	64	Note 2
6	USINT	Unsigned short integer	8	Note 3
7	UINT	Unsigned integer	16	Note 3
8	UDINT	Unsigned double integer	32	Note 3
9	ULINT	Unsigned long integer	64	Note 3
10	REAL	Real numbers	32	Note 4
11	LREAL	Long reals	64	Note 5
12	TIME	Duration	Note 1	Note 6
13	DATE	Date (only)	Note 1	Note 6
14	TIME_OF_DAY or TOD	Time of day (only)	Note 1	Note 6
15	DATE_AND_TIME or DT	Date and time of day	Note 1	Note 6
16	STRING	Variable-length character string	Note 1	Note 7
17	BYTE	Bit string of length 8	8	Note 7
18	WORD	Bit string of length 16	16	Note 7
19	DWORD	Bit string of length 32	32	Note 7
20	LWORD	Bit string of length 64	64	Note 7

NOTES

1 The length of these data elements is implementation-dependent.

2 The range of values for variables of this data type is from -(2**(Bits-1)) to (2**(Bits-1))-1.

3 The range of values for variables of this data type is from 0 to (2**Bits)-1.

4 The range of values for variables of this data type shall be as defined in IEC 559 for the basic single width floating-point format.

5 The range of values for variables of this data type shall be as defined in IEC 559 for the basic double width floating-point format.

6 The range of values for variables of this data type is implementation-dependent.

7 A numeric range of values does not apply to this data type.

8 The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords FALSE and TRUE, respectively.

2.3.2 *Generic data types*

In addition to the data types in table 10, the hierarchy of generic data types shown in table 11 shall be used as defined in 2.5.1.4 in the specification of overloaded inputs and outputs of standard functions and function blocks. Generic data types are identified by the prefix "ANY".

Tableau 11 – Hiérarchie des types de données génériques

ANY ANY_NUM ANY_REAL LREAL REAL ANY_INT LINT, DINT, INT, SINT ULINT, UDINT, UINT, USINT ANY_BIT LWORD, DWORD, WORD, BYTE, BOOL STRING ANY_DATE DATE_AND_TIME DATE TIME_OF_DAY TIME Dérivés (voir notes)
NOTES 1 Les types de données génériques ne doivent pas être utilisés dans des unités d'organisation de programmes déclarées par l'utilisateur, telles que spécifiées en 2.5. 2 Le type générique d'un type dérivé à partir d'une *sous-étendue* (caractéristique 3 du tableau 12) doit être ANY_INT. 3 Le type générique d'un type *directement dérivé* (caractéristique 1 du tableau 12) doit être identique au type générique du type élémentaire dont il est dérivé. 4 Le type générique de tous les autres types dérivés définis au tableau 12 doit être ANY.

2.3.3 *Types de données dérivés*

2.3.3.1 *Déclaration*

Les types de données dérivés (c'est-à-dire, spécifiés par l'utilisateur ou par le fabricant) peuvent être déclarés à l'aide de la construction littérale TYPE...END_TYPE indiquée au tableau 12. Ces types de données dérivés peuvent être utilisés, en plus des types de données élémentaires définis au paragraphe 2.3.1, dans des déclarations de variables telles que celles spécifiées en 2.4.3.

Une déclaration de type de donnée *énuméré* spécifie que la valeur de tout élément de donnée de ce type ne peut prendre qu'une seule des valeurs données dans la liste d'identificateurs associée, comme l'illustre le tableau 12.

Une déclaration de *sous-étendue* spécifie que la valeur de tout élément de donnée de ce type ne peut prendre que des valeurs comprises entre les limites supérieure et inférieure (incluses) spécifiées au tableau 12.

Une déclaration de structure (STRUCT) spécifie que des éléments de données de ce type doivent contenir des sous-éléments de types spécifiés auxquels il est possible d'accéder par l'intermédiaire des noms spécifiés. Par exemple, un élément de type de donnée ANALOG_CHANNEL_CONFIGURATION, tel que déclaré au tableau 12 doit comporter un sous-élément d'ETENDUE (RANGE) du type ANALOG_SIGNAL_RANGE, un sous-élément MIN_SCALE de type ANALOG_DATA, et un élément MAX_SCALE de type ANALOG_DATA.

Table 11 – Hierarchy of generic data types

ANY ANY_NUM ANY_REAL LREAL REAL ANY_INT LINT, DINT, INT, SINT ULINT, UDINT, UINT, USINT ANY_BIT LWORD, DWORD, WORD, BYTE, BOOL STRING ANY_DATE DATE_AND_TIME DATE TIME_OF_DAY TIME Derived (see notes)
NOTES 1 Generic data types shall not be used in user-declared program organization units as defined in 2.5. 2 The generic type of a *subrange* derived type (feature 3 of table 12) shall be ANY_INT. 3 The generic type of a *directly* derived type (feature 1 of table 12) shall be the same as the generic type of the elementary type from which it is derived. 4 The generic type of all other derived types defined in table 12 shall be ANY.

2.3.3 *Derived data types*

2.3.3.1 *Declaration*

Derived (i.e., user- or manufacturer-specified) data types can be declared using the TYPE...END_TYPE textual construction shown in table 12. These derived data types can then be used, in addition to the elementary data types defined in 2.3.1, in variable declarations as defined in 2.4.3.

An *enumerated* data type declaration specifies that the value of any data element of that type can only take on one of the values given in the associated list of identifiers, as illustrated in table 12.

A *subrange* declaration specifies that the value of any data element of that type can only take on values between and including the specified upper and lower limits, as illustrated in table 12.

A STRUCT declaration specifies that data elements of that type shall contain sub-elements of specified types which can be accessed by the specified names. For instance, an element of data type ANALOG_CHANNEL_CONFIGURATION as declared in table 12 will contain a RANGE sub-element of type ANALOG_SIGNAL_RANGE, a MIN_SCALE sub-element of type ANALOG_DATA, and a MAX_SCALE element of type ANALOG_DATA.

Toute déclaration de tableau (ARRAY) spécifie qu'une quantité suffisante de mémoire de données doit être affectée à chaque élément de ce type, pour mémoriser toutes les données qui peuvent être indexées par la(les) sous-étendue(s) d'indices. Ainsi, tout élément de type ANALOG_16_INPUT_CONFIGURATION, tel qu'indiqué au tableau 12 contient (parmi d'autres éléments) une quantité suffisante de mémoire pour 16 éléments CHANNEL de type ANALOG_CHANNEL_CONFIGURATION. Les mécanismes permettant d'accéder aux éléments du tableau sont définis en 2.4.1.2.

2.3.3.2 *Initialisation*

La valeur initiale par défaut d'un type de donnée *énuméré* doit être le premier identificateur dans la liste d'énumérations associée, sauf indication contraire spécifiée par l'opérateur d'affectation ":=". Par exemple, comme l'illustrent les tableaux 12, les valeurs initiales par défaut des éléments de types de données ANALOG_SIGNAL_TYPE et ANALOG_SIGNAL_RANGE sont respectivement SINGLE_ENDED et UNIPOLAR_1_5V.

En ce qui concerne les types de données avec *sous-étendues,* les valeurs initiales par défaut doivent constituer la première limite (inférieure) de l'étendue, sauf indication contraire spécifiée par un opérateur d'affectation. Par exemple, comme l'indique le tableau 12, la valeur initiale par défaut des éléments de type ANALOG_DATA est -4095, alors que la valeur initiale par défaut pour le sous-élément FILTER_PARAMETER des éléments de type ANALOG_16_INPUT_CONFIGURATION est zéro. Par contre, la valeur initiale par défaut d'éléments de type ANALOG_DATA2, tels qu'indiqués au tableau 14 est zéro.

Pour d'autres types de données dérivés, les valeurs initiales par défaut, sauf indication contraire spécifiée par l'utilisation d'un opérateur d'affectation ":=" dans la déclaration de TYPE, doivent être les valeurs initiales par défaut des types de données élémentaires sous-jacents, définis au tableau 13. D'autres exemples d'utilisation de l'opérateur d'affectation pour l'initialisation sont donnés en 2.4.2.

La longueur maximale par défaut des éléments de type CORDON doit être une valeur propre à une application, sauf indication contraire spécifiée par une longueur maximale entre parenthèses (qui ne doit pas dépasser la valeur par défaut propre à l'application concernée) dans la déclaration associée. Par exemple, si le type STR10 est déclaré par

TYPE STR10: STRING(10) := 'ABCDEF' ; END_TYPE

la longueur maximale, la valeur initiale par défaut et la longueur initiale par défaut des éléments de données de type STR10 sont respectivement: 10 caractères, 'ABCDEF' et 6 caractères.

An ARRAY declaration specifies that a sufficient amount of data storage shall be allocated for each element of that type to store all the data which can be indexed by the specified index subrange(s). Thus, any element of type ANALOG_16_INPUT_CONFIGURATION as shown in table 12 contains (among other elements) sufficient storage for 16 CHANNEL elements of type ANALOG_ CHANNEL_CONFIGURATION. Mechanisms for access to array elements are defined in 2.4.1.2.

2.3.3.2 *Initialization*

The default initial value of an *enumerated* data type shall be the first identifier in the associated enumeration list, or a value specified by the assignment operator ":=". For instance, as shown in tables 12 and 14, the default initial values of elements of data types ANALOG_SIGNAL_TYPE and ANALOG_SIGNAL_RANGE are SINGLE_ENDED and UNIPOLAR_1_5V, respectively.

For data types with *subranges*, the default initial values shall be the first (lower) limit of the subrange, unless otherwise specified by an assignment operator. For instance, as declared in table 12, the default initial value of elements of type ANALOG_DATA is 4095, while the default initial value for the FILTER_PARAMETER sub-element of elements of type ANALOG_16_INPUT_CONFIGURATION is zero. In contrast, the default initial value of elements of type ANALOG_DATAZ as declared in table 14 is zero.

For other derived data types, the default initial values, unless specified otherwise by the use of the assignment operator ":=" in the TYPE declaration, shall be the default initial values of the underlying elementary data types as defined in table 13. Further examples of the use of the assignment operator for initialization are given in 2.4.2.

The default maximum length of elements of type STRING shall be an implementation-dependent value unless specified otherwise by a parenthesized maximum length (which shall not exceed the implementation-dependent default value) in the associated declaration. For example, if type STR10 is declared by

```
TYPE STR10 : STRING(10) := 'ABCDEF' ; END_TYPE
```

the maximum length, default initial value, and default initial length of data elements of type STR10 are 10 characters, 'ABCDEF', and 6 characters, respectively.

Tableau 12 – Caractéristiques des déclarations de types de données

N°	Caractéristique/exemple littéral
1	Dérivation directe, à partir de types élémentaires, par exemple: TYPE R : REAL ; END_TYPE
2	Types de données énumérés, par exemple: TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE
3	Types de données sous-étendue, par exemple: TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE
4	Types de données tableau, par exemple: TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE
5	Types de données structurés, par exemple: TYPE ANALOG_CHANNEL_CONFIGURATION : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA ; MAX_SCALE : ANALOG_DATA ; END_STRUCT ; ANALOG_16_INPUT_CONFIGURATION : STRUCT SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ; FILTER_PARAMETER : SINT (0..99) ; CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION ; END_STRUCT ; END_TYPE
NOTE - En ce qui concerne les exemples d'utilisation de ces types dans des déclarations de variables, se reporter en 2.3.3.3 et 2.4.1.2, et voir tableau 17.	

Tableau 13 – Valeurs initiales par défaut

Type(s) de donnée(s)	Valeur initiale
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	''(le cordon vide)

Table 12 – Data type declaration feature

No.	Feature/textual example
1	Direct derivation from elementary types, e.g.: TYPE R : REAL ; END_TYPE
2	Enumerated data types, e.g: TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE
3	Subrange data types, e.g: TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE
4	Array data types, e.g.: TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE
5	Structured data types, e.g.: TYPE ANALOG_CHANNEL_CONFIGURATION : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA ; MAX_SCALE : ANALOG_DATA ; END_STRUCT ; ANALOG_16_INPUT_CONFIGURATION : STRUCT SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ; FILTER_PARAMETER : SINT (0..99) ; CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION ; END_STRUCT ; END_TYPE
NOTE - For examples of the use of these types in variable declarations, see 2.3.3.3, 2.4.1.2, and table 17.	

Table 13 – Default initial values

Data type(s)	Initial value
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	''(the empty string)

Tableau 14 – Caractéristiques de déclaration d'une valeur initiale de type de donnée

N°	Caractéristique/exemple littéral
1	Initialisation de types directement dérivés; par exemple: TYPE FREQ : REAL :- 50.0 ; END_TYPE
2	Initialisation des types de données énumérés; par exemple; TYPE ANALOG_SIGNAL_RANGE : (BIPOLAR_10V, (* –10 à +10 Vc.c. *) UNIPOLAR_10V, (* 0 à +10 Vc.c. *) UNIPOLAR_1_5V, (* +1 à +5 Vc.c. *) UNIPOLAR_0_5V, (* 0 à +5 Vc.c. *) UNIPOLAR_4_20_MA, (* +4 à +20 mAc.c. *) UNIPOLAR_0_20_MA, (* 0 à +20 mAc.c. *)) := UNIPOLAR_1_5V ; END_TYPE
3	Initialisation de types de données sous-étendue; par exemple: TYPE ANALOG_DATAZ : INT (-4095..4095) := 0 ; END_TYPE
4	Initialisation de types de données tableau; par exemple: TYPE ANALOG_16_INPUT_DATAI : ARRAY [1..16] OF ANALOG_DATA := 8(–4095), 8(4095) ; END_TYPE
5	Initialisation d'éléments de types de données structurés; par exemple: TYPE ANALOG_CHANNEL_CONFIGURATION1 : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA := –4095 ; MAX_SCALE : ANALOG_DATA := 4095 ; END_STRUCT ; END_TYPE
6	Initialisation de types de données structurés dérivés; par exemple: TYPE ANALOG_CHANNEL_CONFIGZ : ANALOG_CHANNEL_CONFIGURATION1 (MIN_SCALE := 0, MAX_SCALE := 9999) ; END_TYPE

2.3.3.3 *Utilisation*

L'utilisation de variables qui ont été déclarées (conformément à 2.4.3.1) comme étant de types de données dérivés, doit être conforme aux règles suivantes:

1) Une variable à un seul élément, telle que définie en 2.4.1.1, d'un type dérivé, peut être utilisée partout où une variable de ses types "apparentés" peut être utilisée; par exemple: des variables des types R et PI, telles qu'indiquées aux tableaux 12 et 14, peuvent être utilisées partout où une variable de type REAL peut être utilisée, et des variables de type ANALOG_DATA peuvent être utilisées partout où une variable de type INT peut être utilisée.

Cette règle peut être appliquée de façon récursive. Par exemple, si l'on considère les déclarations ci-dessous, la variable R3 du type R2 peut être utilisée partout où une variable de type REAL peut être utilisée:

```
TYPE R1 : REAL := 1.0 ; END_TYPE
TYPE R2 : R1 ; END_TYPE
VAR R3 : R2 ; END_VAR
```

Table 14 – Data type initial value declaration features

No.	Feature/textual example
1	Initialization of directly derived types, e.g.: TYPE FREQ : REAL :- 50.0 ; END_TYPE
2	Initialization of enumerated data types, e.g: TYPE ANALOG_SIGNAL_RANGE : (BIPOLAR_10V, (* –10 to +10 VDC *) UNIPOLAR_10V, (* 0 to +10 VDC *) UNIPOLAR_1_5V, (* +1 to +5 VDC *) UNIPOLAR_0_5V, (* 0 to +5 VDC *) UNIPOLAR_4_20_MA, (* +4 to +20 mADC *) UNIPOLAR_0_20_MA, (* 0 to +20 mADC *)) := UNIPOLAR_1_5V ; END_TYPE
3	Initialization of subrange data types, e.g.: TYPE ANALOG_DATAZ : INT (-4095..4095) := 0 ; END_TYPE
4	Initialization of array data types, e.g.: TYPE ANALOG_16_INPUT_DATAI : ARRAY [1..16] OF ANALOG_DATA := 8(–4095), 8(4095) ; END_TYPE
5	Initialization of structured data type elements, e.g.: TYPE ANALOG_CHANNEL_CONFIGURATIONI : STRUCT RANGE : ANALOG_SIGNAL_RANGE ; MIN_SCALE : ANALOG_DATA := –4095 ; MAX_SCALE : ANALOG_DATA := 4095 ; END_STRUCT ; END_TYPE
6	Initialization of derived structured data types, e.g.: TYPE ANALOG_CHANNEL_CONFIGZ : ANALOG_CHANNEL_CONFIGURATIONI (MIN_SCALE := 0, MAX_SCALE := 9999) ; END_TYPE

2.3.3.3 *Usage*

The usage of variables which are declared (as defined in 2.4.3.1) to be of derived data types shall conform to the following rules:

1) A single-element variable, as defined in 2.4.1.1, of a derived type, can be used anywhere that a variable of its "parent's" type can be used, e.g. variables of the types R and PI as shown in tables 12 and 14 can be used anywhere that a variable of type REAL could be used, and variables of type ANALOG_DATA can be used anywhere that a variable of type INT could be used.

This rule can be applied recursively. For example, given the declarations below, the variable R3 of type R2 can be used anywhere a variable of type REAL can be used:

```
TYPE R1 : REAL := 1.0 ; END_TYPE
TYPE R2 : R1 ; END_TYPE
VAR R3 : R2 ; END_VAR
```

2) Un élément d'une variable à plusieurs éléments, tel que défini en 2.4.1.2, peut être utilisé partout où le type "apparenté" peut être utilisé; par exemple: si l'on considère la déclaration de ANALOG_16_INPUT_DATA dans le tableau 12 et la déclaration

VAR INS : ANALOG_16_INPUT_DATA ; END_VAR

les variables INS[1] à INS[16] peuvent être utilisées partout où une variable de type INT peut être utilisée.

Cette règle peut être également appliquée de façon récursive; par exemple: si l'on considère les déclarations de ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION, et ANALOG_DATA dans le tableau 12, et la déclaration

VAR CONF : ANALOG_16_INPUT_CONFIGURATION ; END_VAR

la variable CONF.CHANNEL[2].MIN_SCALE peut être utilisée partout où une variable de type INT peut être utilisée.

2.4 *Variables*

Contrairement aux représentations externes de données, décrites en 2.2, les *variables* fournissent un moyen permettant d'identifier des objets de données dont le contenu peut varier, comme par exemple des données associées aux entrées, aux sorties, ou à la mémoire de l'automate programmable. Une variable peut être déclarée comme appartenant à l'un des types élémentaires définis en 2.3.1 ou comme appartenant aux types dérivés, qui sont déclarés conformément à 2.3.3.1.

2.4.1 *Représentation*

2.4.1.1 *Variables à un seul élément*

Une variable à *un seul élément* est définie comme une variable qui représente une valeur mono-élément de l'un des types élémentaires définis en 2.3.1; un type dérivé d'énumération ou de sous-étendue tel que défini en 2.3.3.1; ou un type dérivé dont "la parenté", telle que définie de façon récursive en 2.3.3.3, est rapportable à un type élémentaire, d'énumération ou de sous-étendue. Le présent paragraphe définit le moyen permettant de représenter *symboliquement* de telles variables, ou la manière qui représente *directement* l'association de la donnée élémentaire avec les emplacements physiques ou logiques dans la structure des entrées, des sorties ou de la mémoire des automates programmables.

Les identificateurs, tels que définis en 2.1.2, doivent servir à la représentation symbolique de variables.

La représentation directe d'une variable à un seul élément (mono-élément) doit être assurée par un symbole spécial formé par l'enchaînement d'un signe de pourcentage "%" (position 2/5 dans la table de code ISO/IEC 646), un *préfixe d'emplacement* et un *préfixe de taille* conformes au tableau 15, et un ou plusieurs entiers non signés, séparés par des symboles (.).

Des exemples de variables directement représentées sont indiqués ci-dessous:

%QX75 et %Q75	Bit de sortie 75
%IW215	Emplacement du mot d'entrée 215
%QB7	Emplacement de l'octet de sortie 7
%MD48	Double mot à l'emplacement en mémoire 48
%IW2.5.7.1	Voir explication ci-après

2) An element of a multi-element variable, as defined in 2.4.1.2, can be used anywhere the "parent" type can be used, e.g., given the declaration of ANALOG_16_INPUT_DATA in table 12 and the declaration

VAR INS : ANALOG_16_INPUT_DATA ; END_VAR

the variables INS[1] through INS[16] can be used anywhere that a variable of type INT could be used.

This rule can also be applied recursively, e.g., given the declarations of ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION, and ANALOG_DATA in table 12 and the declaration

VAR CONF : ANALOG_16_INPUT_CONFIGURATION ; END_VAR

the variable CONF.CHANNEL[2].MIN_SCALE can be used anywhere that a variable of type INT could be used.

2.4 *Variables*

In contrast to the external representations of data described in 2.2, *variables* provide a means of identifying data objects whose contents may change, e.g., data associated with the inputs, outputs, or memory of the programmable controller. A variable can be declared to be one of the elementary types defined in 2.3.1, or one of the derived types which are declared as defined in 2.3.3.1.

2.4.1 *Representation*

2.4.1.1 *Single-element variables*

A *single-element variable* is defined as a variable which represents a single data element of one of the elementary types defined in 2.3.1; a derived enumeration or subrange type as defined in 2.3.3.1; or a derived type whose "parentage", as defined recursively in 2.3.3.3, is traceable to an elementary, enumeration or subrange type. This subclause defines the means of representing such variables *symbolically*, or alternatively in a manner which *directly* represents the association of the data element with physical or logical locations in the programmable controller's input, output, or memory structure.

Identifiers, as defined in 2.1.2, shall be used for symbolic representation of variables.

Direct representation of a single-element variable shall be provided by a special symbol formed by the concatenation of the percent sign "%" (position 2/5 in the ISO/IEC 646 code table), a *location prefix* and a *size prefix* from table 15, and one or more unsigned integers, separated by periods (.)

Examples of directly represented variables are:

%QX75 and %Q75	Output bit 75
%IW215	Input word location 215
%QB7	Output byte location 7
%MD48	Double word at memory location 48
%IW2.5.7.1	See explanation below

Le fabricant doit spécifier la correspondance entre la représentation directe d'une variable et l'emplacement physique ou logique de l'élément adressé en mémoire, en entrée ou en sortie. Lorsqu'une représentation directe est étendue avec des zones d'entiers séparées par des points, elle doit être interprétée comme une adresse *hiérarchique* physique ou logique, où la zone la plus à gauche représente le niveau le plus élevé de la hiérarchie, les niveaux inférieurs apparaissant successivement à droite. Par exemple, la variable %IW2.5.7.1 peut représenter la première "voie" (mot) du septième "module" dans le cinquième "tiroir" du second "bus d'entrée/sortie" d'un système d'automate programmable.

L'utilisation d'un adressage hiérarchique pour permettre à un programme dans un système d'automates programmables d'accéder à des données dans un autre automate programmable, doit être considérée comme une extension de langage.

L'utilisation de variables directement représentées est uniquement autorisée dans des *programmes,* tels que définis en 2.5.3, et dans des *configurations* et des *ressources,* telles que décrites en 2.7.1. Le nombre maximal de niveaux d'adressage hiérarchique est un paramètre propre à une application.

Tableau 15 – Caractéristiques des préfixes d'emplacement et de taille pour des variables représentées directement

N°	Préfixe	Signification
1	I	Emplacement d'entrée
2	Q	Emplacement de sortie
3	M	Emplacement de mémoire
4	X	Taille d'un seul bit
5	Aucun	Taille d'un seul bit
6	B	Taille d'un octet (8 bits)
7	W	Taille d'un mot (16 bits)
8	D	Taille d'un double mot (32 bits)
9	L	Taille d'un mot long (Quad) (64 bits)
NOTES 1 Sauf déclaration contraire, le type de donnée d'une variable directement adressable, de la taille d'un "seul bit" doit être BOOL. 2 Les organismes nationaux de normalisation peuvent publier des tables de traduction de ces préfixes.		

2.4.1.2 *Variables à plusieurs éléments*

Les types de variables à *plusieurs éléments* (multi-éléments), définis dans la présente norme, sont des *tableaux* et des *structures.*

Un *tableau* est un ensemble d'éléments de données appartenant au même type de donnée, référencé par un ou plusieurs *indices* mis entre crochets et séparés par des virgules. Un indice doit être une expression donnant une valeur correspondant à l'un des sous-types du type générique ANY_INT, tel qu'illustré dans le tableau 11.

On trouvera ci-dessous un exemple d'utilisation de variables de tableau, exprimées en langage ST tel qu'il a été défini en 3.3:

```
OUTARY[%MB6,SYM] := INARY[0] + INARY[7] – INARY[%MB6] * %IW62 ;
```

The manufacturer shall specify the correspondence between the direct representation of a variable and the physical or logical location of the addressed item in memory, input or output. When a direct representation is extended with additional integer fields separated by periods, it shall be interpreted as a *hierarchical* physical or logical address with the leftmost field representing the highest level of the hierarchy, with successively lower levels appearing to the right. For instance, the variable %IW2.5.7.1 may represent the first "channel" (word) of the seventh "module" in the fifth "rack" of the second "I/O bus" of a programmable controller system.

The use of hierarchical addressing to permit a program in one programmable controller system to access data in another programmable controller shall be considered a language extension.

The use of directly represented variables is only permitted in *programs*, as defined in 2.5.3, and in *configurations* and *resources* as defined in 2.7.1. The maximum number of levels of hierarchical addressing is an implementation-dependent parameter.

Table 15 – Location and size prefix features for directly represented variables

No.	Prefix	Meaning
1	I	Input location
2	Q	Output location
3	M	Memory location
4	X	Single bit size
5	None	Single bit size
6	B	Byte (8 bits) size
7	W	Word (16 bits) size
8	D	Double word (32 bits) size
9	L	Long (quad) word (64 bits) size

NOTES

1 Unless otherwise declared, the data type of a directly addressed variable of "single bit" size shall be BOOL.

2 National standards organizations can publish tables of translations of these prefixes.

2.4.1.2 *Multi-element variables*

The *multi-element variable* types defined in this standard are *arrays* and *structures.*

An *array* is a collection of data elements of the same data type referenced by one or more *subscripts* enclosed in brackets and separated by commas. A subscript shall be an expression yielding a value corresponding to one of the sub-types of generic type ANY_INT as defined in table 11.

An example of the use of array variables in the ST language as defined in 3.3 is:

```
OUTARY[%MB6,SYM] := INARY[0] + INARY[7] – INARY[%MB6] * %IW62 ;
```

Le nombre maximal d'indices, et l'étendue maximale des valeurs d'indices, qui peuvent être utilisés pour accéder à des variables de tableau, est un paramètre propre à une application donnée.

Une – *variable structurée* est une variable qui est déclarée comme appartenant à un type qui a été précédemment spécifié comme une *structure de données,* c'est-à-dire un type de donnée composé d'un ensemble d'éléments nommés.

Un élément d'une variable structurée doit être représenté par au moins deux identificateurs ou accès aux tableaux, séparés par un symbole point unique (.). Le premier identificateur représente le nom de l'élément structuré, et les identificateurs suivants représentent la séquence des noms de composants permettant d'accéder à un élément de donnée particulier dans la structure de données.

Par exemple, si la variable MODULE_5_CONFIG a été déclarée comme étant du type ANALOG_16_INPUT_CONFIGURATION, tel qu'illustré dans le tableau 12, les instructions suivantes, exprimées dans le langage ST défini en 3.3, doivent entraîner l'affectation de la valeur SINGLE_ENDED à l'élément SIGNAL_TYPE de la variable MODULE_5_CONFIG, alors que la valeur BIPOLAR_10V devrait être affectée au sous-élément RANGE du cinquième élément CHANNEL de MODULE_5_CONFIG:

```
MODULE_5_CONFIG.SIGNAL_TYPE := SINGLE_ENDED ;
MODULE_5_CONFIG.CHANNEL[5].RANGE := BIPOLAR_10V ;
```

Le nombre maximal de niveaux de l'affectation d'un élément de structure est un paramètre propre à une application.

2.4.2 *Initialisation*

Lorsqu'un élément de configuration (*ressource* ou *configuration)* est "lancé" conformément au 1.4.1, chacune des variables associées à l'élément de configuration et à son *programme* peut prendre une des valeurs initiales suivantes:

- la valeur de la variable lorsque l'élément de configuration a été "arrêté" (une valeur *retenue);*
- une valeur initiale spécifiée par l'utilisateur;
- la valeur initiale par défaut relative au type de donnée associé de la variable.

L'utilisateur peut déclarer qu'une valeur doit être *non-volatile,* à l'aide du qualificatif RETAIN spécifié au tableau 16, lorsque cette caractéristique est acceptée par la mise en oeuvre.

La valeur initiale d'une variable, au moment du lancement de son élément de configuration associé, doit être déterminée conformément aux règles suivantes:

1) Si l'opération de démarrage est une "reprise à chaud", telle que définie dans la CEI 1131-1, les valeurs initiales des variables *non-volatiles* doivent être leurs valeurs *retenues,* telles que définies ci-dessus.

The maximum number of subscripts, and the maximum range of subscript values, which may be used to access array variables is an implementation-dependent parameter.

A *structured variable* is a variable which is declared to be of a type which has previously been specified to be a *data structure*, i.e., a data type consisting of a collection of named elements.

An element of a structured variable shall be represented by two or more identifiers or array accesses separated by single periods (.). The first identifier represents the name of the structured element, and subsequent identifiers represent the sequence of component names to access the particular data element within the data structure.

For instance, if the variable MODULE_5_CONFIG has been declared to be of type ANALOG_16_INPUT_CONFIGURATION as shown in table 12, the following statements in the ST language defined in 3.3 would cause the value SINGLE_ENDED to be assigned to the element SIGNAL_TYPE of the variable MODULE_5_CONFIG, while the value BIPOLAR_10V would be assigned to the RANGE sub-element of the fifth CHANNEL element of MODULE_5_CONFIG:

```
MODULE_5_CONFIG.SIGNAL_TYPE := SINGLE_ENDED ;
MODULE_5_CONFIG.CHANNEL[5].RANGE := BIPOLAR_10V
```

The maximum number of levels of structure element addressing is an implementation-dependent parameter.

2.4.2 *Initialization*

When a configuration element (*resource* or *configuration*) is "started" as defined in 1.4.1, each of the variables associated with the configuration element and its *programs* can take on one of the following initial values:

- the value the variable had when the configuration element was "stopped" (a *retained* value);
- a user-specified initial value;
- the default initial value for the variable's associated data type.

The user can declare that a variable is to be *retentive* by using the RETAIN qualifier specified in table 16, when this feature is supported by the implementation.

The initial value of a variable upon starting of its associated configuration element shall be determined according to the following rules:

1) If the starting operation is a "warm restart" as defined in IEC 1131-1, the initial values of *retentive* variables shall be their *retained* values as defined above.

2) Si l'opération est une "reprise à froid", telle que définie dans la CEI 1131-1, les valeurs initiales de variables *non volatiles* doivent être les valeurs initiales spécifiées par l'utilisateur, ou la valeur par défaut, comme cela a été défini en 2.3.3.2, pour le type de donnée associé de toute variable pour laquelle aucune valeur initiale n'est spécifiée par l'utilisateur.

3) Les variables non retenues doivent être initialisées aux valeurs initiales spécifiées par l'utilisateur, ou à la valeur par défaut, comme cela a été défini en 2.3.3.2, pour le type de donnée associé de toute variable pour laquelle aucune valeur initiale n'est spécifiée par l'utilisateur.

4) Des variables qui représentent des *entrées* du *système d'automate* programmable, comme cela est défini dans la CEI 1131-1, doivent être initialisées conformément à la mise en oeuvre considérée.

2.4.3 *Déclaration*

Le début de chaque déclaration de type d'unité d'organisation de programme d'automate programmable (c'est-à-dire, chaque déclaration de *programme,* de *fonction,* ou de *bloc fonctionnel,* comme cela est défini en 2.5), doit comporter au moins une *partie de déclaration* qui spécifie les types (et, si nécessaire, l'emplacement physique ou logique) des variables utilisées dans l'unité d'organisation. Cette partie de déclaration doit avoir la forme littérale de l'un des mots clés VAR, VAR_INPUT, ou VAR_OUTPUT, comme cela est défini au tableau 16, suivie dans le cas de VAR, de zéro ou une apparition du qualificatif RETAIN ou du qualificatif CONSTANT et, dans le cas de VAR_OUTPUT, du qualificatif RETAIN, suivi d'une ou de plusieurs déclarations séparées par des points-virgules. Chaque déclaration doit être terminée par le mot clé END_VAR. Lorsqu'un automate programmable accepte la déclaration, faite par l'utilisateur, de valeurs initiales relatives à des variables, cette déclaration doit être accomplie dans la (les) partie(s) réservée(s) à la déclaration, telles que définies au présent paragraphe.

Le *champ d'application* (étendue de validité) des déclarations contenues dans la partie déclaration, doit être *local* à l'unité d'organisation de programme qui contient la partie déclaration. Cela veut dire que les variables déclarées ne doivent pas être accessibles à d'autres unités d'organisation de programmes, sauf par un paramètre explicite passant par des variables qui ont été déclarées comme des *entrées* ou des *sorties* de ces unités. La seule exception à cette règle concerne le cas de variables qui ont été déclarées comme des variables *globales,* telles que définies en 2.7.1. De telles variables ne sont accessibles à une unité d'organisation de programme que par l'intermédiaire d'une déclaration VAR_EXTERNAL. Le type d'une variable déclarée dans un bloc VAR_EXTERNAL doit être en accord avec le type déclaré dans le bloc VAR_GLOBAL du *programme,* de la *configuration* ou de la *ressource* associé(e).

2) If the operation is a "cold restart" as defined in IEC 1131-1, the initial values of retentive variables shall be the user-specified initial values, or the default value, as defined in 2.3.3.2, for the associated data type of any variable for which no initial value is specified by the user.

3) Non-retained variables shall be initialized to the user-specified initial values, or to the default value, as defined in 2.3.3.2, for the associated data type of any variable for which no initial value is specified by the user.

4) Variables which represent *inputs* of the *programmable controller system* as defined in IEC 1131-1 shall be initialized in an implementation-dependent manner.

2.4.3 *Declaration*

Each programmable controller program organization unit type declaration (i.e., each declaration of a *program, function,* or *function block*, as defined in 2.5) shall contain at its beginning at least one *declaration part* which specifies the types (and, if necessary, the physical or logical location) of the variables used in the organization unit. This declaration part shall have the textual form of one of the keywords VAR, VAR_INPUT, or VAR_OUTPUT as defined in table 16, followed in the case of VAR by zero or one occurrence of the qualifier RETAIN or the qualifier CONSTANT, and in the case of VAR_OUTPUT by zero or one occurrence of the qualifier RETAIN, followed by one or more declarations separated by semicolons and terminated by the keyword END_VAR. When a programmable controller supports the declaration by the user of initial values for variables, this declaration shall be accomplished in the declaration part(s) as defined in this subclause.

The *scope* (range of validity) of the declarations contained in the declaration part shall be *local* to the program organization unit in which the declaration part is contained. That is, the declared variables shall not be accessible to other program organization units except by explicit parameter passing via variables which have been declared as *inputs* or *outputs* of those units. The one exception to this rule is the case of variables which have been declared to be *global*, as defined in 2.7.1. Such variables are only accessible to a program organization unit via a VAR_EXTERNAL declaration. The type of a variable declared in a VAR_EXTERNAL block shall agree with the type declared in the VAR_GLOBAL block of the associated *program, configuration* or *resource.*

Tableau 16 – Mots clés de déclarations de variables

Mot clé	Utilisation de variables
VAR	Interne à l'unité d'organisation
VAR_INPUT	Fournie de l'extérieur, non modifiable dans l'unité d'organisation
VAR_OUTPUT	Fournie par l'unité d'organisation aux entités externes
VAR_IN_OUT	Fournie par des entités externes Peut être modifiée dans l'unité d'organisation NOTE - Des exemples de l'utilisation de ces variables sont donnés dans les figures 11b et 12.
VAR_EXTERNAL	Fournie par la configuration, par l'intermédiaire de VAR_GLOBAL (2.7.1) Peut être modifiée dans l'unité d'organisation
VAR_GLOBAL	Déclaration de variable globale (2.7.1)
VAR_ACCESS	Déclaration de chemin d'accès (2.7.1)
RETAIN	Variables non volatiles (voir texte précédent)
CONSTANT	Constante (variable qui ne peut être modifiée)
AT	Enoncé d'emplacement (voir 2.4.3.1)
NOTE - L'utilisation de ces mots clés est une caractéristique de l'unité d'organisation de programme ou un élément de configuration dans lequel ils sont utilisés; voir 2.5 et 2.7.	

2.4.3.1 *Enoncé de type*

Comme il apparaît au tableau 17, la construction VAR...END_VAR doit être utilisée pour spécifier des types de données ainsi que la non-volatilité de variables directement représentées. Cette construction doit également être utilisée pour spécifier des types de données, la non-volatilité, et (si cela est nécessaire, dans les programmes uniquement) l'emplacement physique ou logique des variables à un seul élément ou à plusieurs éléments, représentées symboliquement. L'utilisation des constructions VAR_INPUT, VAR_OUTPUT et VAR_IN_OUT est définie en 2.5.

L'affectation d'une adresse physique ou logique, à une variable représentée symboliquement, doit être réalisée à l'aide du mot clé AT. Lorsqu'aucune affectation de ce type n'est effectuée, il est nécessaire de prévoir l'affectation automatique de la variable à un emplacement approprié dans la mémoire de l'automate programmable.

Table 16 – Variable declaration keywords

Keyword	Variable usage
VAR	Internal to organization unit
VAR_INPUT	Externally supplied, not modifiable within organization unit
VAR_OUTPUT	Supplied by organization unit to external entities
VAR_IN_OUT	Supplied by external entities Can be modified within organization unit NOTE - Examples of the use of these variables are given in figures 11b and 12
VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL (2.7.1) Can be modified within organization unit
VAR_GLOBAL	Global variable declaration (2.7.1)
VAR_ACCESS	Access path declaration (2.7.1)
RETAIN	Retentive variables (see preceding text)
CONSTANT	Constant (variable cannot be modified)
AT	Location assignment (2.4.3.1)
NOTE - The usage of these keywords is a feature of the program organization unit or configuration element in which they are used; see 2.5 and 2.7.	

2.4.3.1 *Type assignment*

As shown in table 17, the VAR...END_VAR construction shall be used to specify data types and retentivity for directly represented variables. This construction shall also be used to specify data types, retentivity, and (where necessary, in *programs* only) the physical or logical location of symbolically represented single- or multi-element variables. The usage of the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT constructions is defined in 2.5.

The assignment of a physical or logical address to a symbolically represented variable shall be accomplished by the use of the AT keyword. Where no such assignment is made, automatic allocation of the variable to an appropriate location in the programmable controller memory shall be provided.

Tableau 17 – Caractéristiques d'affectation de types de variables

N°	Caractéristique/exemples	
1	Déclaration de variables volatiles, directement représentées	

	VAR
	  AT %IW6.2 : WORD ;
	  AT %MW6 : INT ;
	END_VAR
		Cordon à 16 bits (note 2)
		Entier à 16 bits, valeur initiale = 0

2	Déclaration de variables non volatiles, directement représentées	
	VAR RETAIN AT %QW5 : WORD ; END_VAR	A la reprise à froid, %QW5 sera initialisé à un cordon à 16 bits avec la valeur 0
3	Déclaration d'emplacements de variables symboliques	

	VAR_GLOBAL
	  LIM_SW_5 AT %IX27 : BOOL ;
	
	  CONV_START AT %QX25 : BOOL ;
	
	  TEMPERATURE AT %IW28 : INT ;
	
	END_VAR
		Affecte le bit d'entrée 27 à la variable booléenne LIM_SW_5 (note 2)
		Affecte le bit de sortie 25 à la variable booléenne CONV_START
		Affecte le mot d'entrée 28 à la variable entière TEMPERATURE (note 2)

4	Affectation d'emplacements de tableaux	

	VAR
	  INARY AT %IW6 :
	  ARRAY [0..9] OF INT ;
	END_VAR
		Déclare un tableau de 10 entiers à affecter à des emplacements d'entrée contigus, commençant à %IW6 (note 2)

5	Affectation automatique en mémoire de variables symboliques	

	VAR
	  CONDITION_RED : BOOL ;
	  IBOUNCE : WORD ;
	
	  MYDUB : DWORD ;
	
	  AWORD, BWORD, CWORD : INT ;
	
	  MYSTR : STRING(10) ;
	END_VAR
		Affecte un bit de mémoire à la variable booléenne CONDITION_RED
		Affecte un mot de mémoire à la variable IBOUNCE à chaîne de 16 bits
		Affecte un double mot de mémoire à la variable MYDUB à chaîne de 32 bits
		Affecte 3 mots de mémoire séparés pour les variables entières AWORD, BWORD et CWORD
		Affecte la mémoire qui doit contenir un cordon d'une longueur maximale de 10 caractères. Après l'initialisation, le cordon a une longueur égale à 0 et contient le cordon vide".

6	Déclaration de tableaux	
	VAR THREE : ARRAY[1..5,1..10,1..8] OF INT ; END_VAR	Affecte 400 mots de mémoire pour un tableau tridimensionnel d'entiers
7	Déclaration de tableaux non volatiles	
	VAR RETAIN RTBT : ARRAY[1..2,1..3] OF INT ; END_VAR	Déclare le tableau non-volatile RTBT, avec des valeurs initiales 0 pour tous les éléments lors d'une "reprise à froid".
8	Déclaration de variables structurées	
	VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION ; END_VAR	Déclaration d'une variable de type de donnée dérivé (voir tableau 12)

NOTES:

1 Les caractéristiques 1 à 4 ne peuvent être utilisées que dans des déclarations PROGRAM et VAR_GLOBAL, respectivement définies aux paragraphes 2.5.3 et 2.7.1.

2 L'initialisation des entrées du système est propre à la mise en oeuvre; se reporter au paragraphe 2.4.2.

Table 17 – Variable type assignment features

No.	Feature/examples	
1	Declaration of directly represented, non-retentive variable	

	VAR
	  AT %IW6.2 : WORD ;
	  AT %MW6 : INT ;
	END_VAR
		16-bit string (note 2)
		16-bit integer, initial value = 0

2	Declaration of directly represented retentive variables	
	VAR RETAIN AT %QW5 : WORD ; END_VAR	At cold restart, %QW5 will be initialized to a 16-bit string with value 0
3	Declaration of locations of symbolic variables	

	VAR_GLOBAL
	  LIM_SW_5 AT %IX27 : BOOL ;
	
	  CONV_START AT %QX25 : BOOL ;
	
	  TEMPERATURE AT %IW28 : INT ;
	
	END_VAR
		Assigns input bit 27 to the Boolean variable LIM_SW_5 (note 2)
		Assigns output bit 25 to the Boolean variable CONV_START
		Assigns input word 28 to the integer variable TEMPERATURE (note 2)

4	Array location assignment	

	VAR
	  INARY AT %IW6 :
	  ARRAY [0..9] OF INT ;
	END_VAR
		Declares an array of 10 integers to be allocated to contiguous input locations starting at %IW6 (note 2)

5	Automatic memory allocation of symbolic variables	

	VAR
	  CONDITION_RED : BOOL ;
	  IBOUNCE : WORD ;
	
	  MYDUB : DWORD ;
	
	  AWORD, BWORD, CWORD : INT ;
	
	  MYSTR : STRING(10) ;
	END_VAR
		Allocates a memory bit to the Boolean variable CONDITION_RED
		Allocates a memory word to the 16-bit string variable IBOUNCE.
		Allocates a double memory word to the 32-bit string variable MYDUB.
		Allocates 3 separate memory words for the integer variables AWORD, BWORD and CWORD
		Allocates memory to contain a string with a maximum length of 10 characters. After initialization, the string has length 0 and contains the empty string''.

6	Array declaration	
	VAR THREE : ARRAY[1..5,1..10,1..8] OF INT ; END_VAR	Allocates 400 memory words for a three-dimensional array of integers
7	Retentive array declaration	
	VAR RETAIN RTBT : ARRAY[1..2,1..3] OF INT ; END_VAR	Declares retentive array RTBT with "cold restart" initial values of 0 for all elements
8	Declaration of structured variables	
	VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION ; END_VAR	Declaration of a variable of derived data type (see table 12)

NOTES:

1 Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations, as defined in 2.5.3 and 2.7.1 respectively.

2 Initialization of system inputs is implementation-dependent; see 2.4.2.

2.4.3.2 *Affectation de valeurs initiales*

La construction VAR...END_VAR indiquée au tableau 18 doit être utilisée pour spécifier les valeurs initiales de variables directement représentées. Cette construction doit être également utilisée pour affecter les valeurs initiales de variables à un seul élément ou à plusieurs éléments, représentées symboliquement (l'utilisation des constructions VAR_INPUT, VAR_OUTPUT et VAR_IN_OUT est définie en 2.5).

Les valeurs initiales ne peuvent être données dans des déclarations VAR_EXTERNAL.

Pendant l'initialisation de tableaux, l'indice le plus à droite d'un tableau doit varier le plus rapidement par rapport au remplissage du tableau à partir de la liste des variables d'initialisation.

Il est possible d'utiliser des parenthèses comme un facteur d'itération dans des listes d'initialisation de tableaux; par exemple, "2(1,2,3)" est équivalent à la séquence d'initialisation "1,2,3,1,2,3".

Si le nombre de valeurs initiales, donné dans la liste d'initialisation, est supérieur au nombre d'entrées du tableau, les valeurs initiales en excès (les plus à droite) doivent être ignorées. Si le nombre de valeurs initiales est inférieur au nombre d'entrées du tableau, les entrées restantes du tableau doivent contenir les valeurs initiales par défaut relatives au type de donnée correspondant. Dans un cas ou dans l'autre, l'utilisateur doit être averti de cette condition, pendant la préparation du programme pour l'exécution.

Lorsqu'une variable est déclarée comme appartenant à un type de donnée dérivé structuré, tel que défini en 2.3.3.1, des valeurs initiales, relatives aux éléments de la variable, peuvent être déclarées dans une liste mise entre parenthèses à la suite de l'identificateur de type de donnée, comme indiqué au tableau 18. Les éléments pour lesquels aucune valeur initiale n'est présente dans la liste de valeurs initiales, doivent avoir les valeurs initiales par défaut déclarées pour les éléments dans la déclaration de type de donnée.

2.4.3.2 *Initial value assignment*

The VAR...END_VAR construction shown in table 18 shall be used to specify initial values of directly represented variables. This construction shall also be used to assign initial values of symbolically represented single- or multi-element variables (the usage of the VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT constructions is defined in 2.5).

Initial values cannot be given in VAR_EXTERNAL declarations.

During initialization of arrays, the rightmost subscript of an array shall vary most rapidly with respect to filling the array from the list of initialization variables.

Parentheses can be used as a repetition factor in array initialization lists, e.g., "2(1,2,3)" is equivalent to the initialization sequence "1,2,3,1,2,3".

If the number of initial values given in the initialization list exceeds the number of array entries, the excess (rightmost) initial values shall be ignored. If the number of initial values is less than the number of array entries, the remaining array entries shall be filled with the default initial values for the corresponding data type. In either case, the user shall be warned of this condition during preparation of the program for execution.

When a variable is declared to be of a derived, structured data type as defined in 2.3.3.1, initial values for the elements of the variable can be declared in a parenthesized list following the data type identifier, as shown in table 18. Elements for which initial values are not listed in the initial value list shall have the default initial values declared for those elements in the data type declaration.

Tableau 18 – Caractéristiques d'affectation de valeurs initiales de variables

N°	Caractéristique/exemples	
1	Initialisation de variables volatiles, directement représentées	
	VAR AT %QX 5.1 : BOOL := 1 ; AT %MW6 : INT := 8 ; END_VAR	Type booléen, valeur initiale =1 Initialise un mot de mémoire en entier 8
2	Initialisation de variables non volatiles, directement représentées	
	VAR RETAIN AT %QW5 : WORD := 16#FF00 ; END_VAR	Lors d'une reprise à froid, les 8 bits les plus significatifs du cordon à 16 bits au niveau du mot de sortie 5 doivent être initialisés à 1 et les 8 bits les moins significatifs doivent être initialisés à 0.
3	Affectation d'emplacements et de valeurs initiales à des variables symboliques	
	VAR VALVE_POS AT %QW28 : INT := 100 ; END_VAR	Affecte le mot de sortie 28 à la variable entière VALVE_POS, avec une valeur initiale de 100
4	Affectation et initialisation d'emplacement de tableaux	
	VAR OUTARY AT %QW6: ARRAY [0..9] OF INT := 10(1) ; END_VAR	Déclare un tableau de 10 entiers à affecter à des emplacements de sortie contigus, commençant à %QW6, chacun ayant une valeur initiale de 1.
5	Initialisation de variables symboliques	
	VAR MYBIT : BOOL := 1 ; OKAY : STRING(10) := 'OK' ; END_VAR	Affecte un bit de mémoire à la variable booléenne MYBIT avec une valeur initiale de 1 Affecte la mémoire qui doit contenir un cordon d'une longueur maximale de 10 caractères. Après l'initialisation, la longueur du cordon est 2 et contient la séquence, à 2 octets, des caractères 'OK' dans le jeu de caractères ISO/IEC 646, dans un ordre qui convient pour l'impression en tant que cordon de caractères.
6	Initialisation de tableau	
	VAR BITS : ARRAY [0..7] OF BOOL := 1,1,0,0,0,1,0,0 ; TBT : ARRAY[1..2,1..3] OF INT : = 1,2,3(4),6 ; END_VAR	Affecte 8 bits de mémoire destinés à contenir des valeurs initiales BITS[0] := 1, BITS [1] := 1, ..., BITS[6] := 0, BITS [7] := 0 Affecte un tableau d'entiers 2 par 3 TBT avec des valeurs initiales TBT[1,1] := 1, TBT[1,2] := 2, TBT[1,3] := 4, TBT[2,1] := 4, TBT[2,2] := 4, TBT[2,3] := 6,
7	Déclaration et initialisation d'un tableau non volatil	
	VAR RETAIN RTBT : ARRAY [1..2,1..3] OF INT : = 1,2,3(4) ; END_VAR	Déclare le tableau non volatil RTBT avec des valeurs initiales "reprise à froid" de: RTBT[1,1] := 1, RTBT[1,2] := 2, RTBT[1,3] := 4, RTBT[2,1] := 4, RTBT[2,2] := 4, RTBT[2,3] := 0
8	Initialisation de variable structurées	
	VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION (SIGNAL_TYPE := DIFFERENTIAL, CHANNEL[5].RANGE := BIPOLAR_10_V, CHANNEL[5].MIN_SCALE := 0, CHANNEL[5].MAX_SCALE := 500) ; END_VAR	Initialisation d'une variable de type de donnée dérivé (Voir tableau 12)
9	Initialisation de constantes	
	VAR CONSTANT PI : REAL := 3.141592 ; END_VAR	
	NOTE - Les caractéristiques 1 à 4 ne peuvent être utilisées que dans des déclarations PROGRAM et VAR_GLOBAL, respectivement définies en 2.5.3 et 2.7.1.	

Table 18 – Variable initial value assignment features

No.	Feature/examples	
1	Initialization of directly represented, non-retentive variables	
	VAR AT %QX 5.1 : BOOL := 1 ; AT %MW6 : INT := 8 ; END_VAR	Boolean type, initial value =1 Initializes a memory word to integer 8
2	Initialization of directly represented retentive variables	
	VAR RETAIN AT %QW5 : WORD := 16#FF00 ; END_VAR	At cold restart, the 8 most significant bits of the 16-bit string at output word 5 are to be initialized to 1 and the 8 least significant bits to 0
3	Location and initial value assignment to symbolic variables	
	VAR VALVE_POS AT %QW28 : INT := 100 ; END_VAR	Assigns output word 28 to the integer variable VALVE_POS, with an initial value of 100
4	Array location assignment and initialization	
	VAR OUTARY AT %QW6: ARRAY [0..9] OF INT := 10(1) ; END_VAR	Declares an array of 10 integers to be allocated to contiguous output locations starting at %QW6, each with an initial value of 1
5	Initialization of symbolic variables	
	VAR MYBIT : BOOL := 1 ; OKAY : STRING(10) := 'OK' ; END_VAR	Allocates a memory bit to the Boolean variable MYBIT with an initial value of 1. Allocates memory to contain a string with a maximum length of 10 characters. After initialization, the string has length 2 and contains the two-byte sequence of characters 'OK' in the ISO/IEC 646 character set, in an order appropriate for printing as a character string.
6	Array initialization	
	VAR BITS : ARRAY [0..7] OF BOOL := 1,1,0,0,0,1,0,0 ; TBT : ARRAY[1..2,1..3] OF INT : = 1,2,3(4),6 ; END_VAR	Allocates 8 memory bits to contain initial values BITS[0] := 1, BITS [1] := 1, ..., BITS[6] := 0, BITS [7] := 0 Allocates a 2-by-3 integer array TBT with initial values TBT[1,1] := 1, TBT[1,2] := 2, TBT[1,3] := 4, TBT[2,1] := 4, TBT[2,2] := 4, TBT[2,3] := 6,
7	Retentive array declaration and initialization	
	VAR RETAIN RTBT : ARRAY [1..2, 1..3] OF INT : = 1,2,3(4) ; END_VAR	Declares retentive array RTBT with "cold restart" initial values of: RTBT[1,1] := 1, RTBT[1,2] := 2, RTBT[1,3] := 4, RTBT[2,1] := 4, RTBT[2,2] := 4, RTBT[2,3] := 0
8	Initialization of structured variables	
	VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION (SIGNAL_TYPE := DIFFERENTIAL, CHANNEL[5].RANGE := BIPOLAR_10_V, CHANNEL[5].MIN_SCALE := 0, CHANNEL[5].MAX_SCALE := 500) ; END_VAR	Initialization of a variable of derived data type (see table 12)
9	Initialization of constants	
	VAR CONSTANT PI : REAL := 3.141592 ; END_VAR	
	NOTE - Features 1 to 4 can only be used in PROGRAM and VAR_GLOBAL declarations, as defined in 2.5.3 and 2.7.1 respectively.	

2.5 *Unités d'organisation de programmes*

Les unités d'organisation de programmes, définies dans cette partie de la CEI 1131 sont la *fonction,* le *bloc fonctionnel* et le *programme.* Ces unités d'organisation de programmes peuvent être fournies par le fabricant, ou programmées par l'utilisateur à l'aide des moyens définis dans cette partie de la norme.

Les unités d'organisation de programmes ne doivent pas être *récursives;* c'est-à-dire que le lancement d'une unité d'organisation de programme ne doit pas entraîner le lancement d'une autre unité d'organisation de programme du même type.

2.5.1 *Fonctions*

Pour les besoins des langages de programmation d'automates programmables, une *fonction* est définie comme une unité d'organisation de programme qui, lorsqu'elle est exécutée, donne exactement un élément de donnée (qui peut prendre plusieurs valeurs, comme par exemple un tableau ou une structure), et dont le lancement peut être utilisé, dans des langages littéraux, comme un opérande dans une expression. Par exemple: les fonctions SIN et COS pourraient être utilisées comme l'illustre la figure 4.

a)

```
Z := SIN(X)*COS(Y) + COS(X)*SIN(Y) ;
```

b)

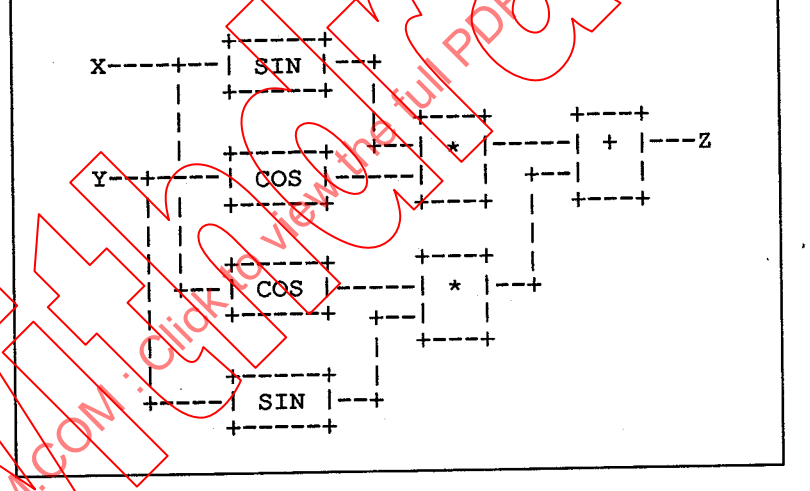


Figure 4 – Exemples d'utilisation de fonctions
a) langage littéral structuré (ST); voir 3.3
b) langage FBD (Schéma en blocs fonctionnels); voir 4.3

Les fonctions ne doivent comporter aucune information concernant les états internes, c'est-à-dire que le lancement d'une fonction dotée des mêmes arguments (paramètres d'entrée) doit toujours donner la même valeur (sortie).

Tout type de fonction qui a déjà été déclaré peut être utilisé dans la déclaration d'une autre unité d'organisation de programme, comme l'illustre la figure 3.

2.5 *Program organization units*

The program organization units defined in this part of IEC 1311 are the *function, function block*, and *program*. These program organization units can be delivered by the manufacturer, or programmed by the user by the means defined in this part of the standard.

Program organization units shall not be *recursive*; that is, the invocation of a program organization unit shall not cause the invocation of another program organization unit of the same type.

2.5.1 *Functions*

For the purposes of programmable controller programming languages, a *function* is defined as a program organization unit which, when executed, yields exactly one data element (which can be multi-valued, e.g., an array or structure), and whose invocation can be used in textual languages as an operand in an expression. For example, the SIN and COS functions could be used as shown in figure 4.

a)

```
Z := SIN(X)*COS(Y) + COS(X)*SIN(Y);
```

b)

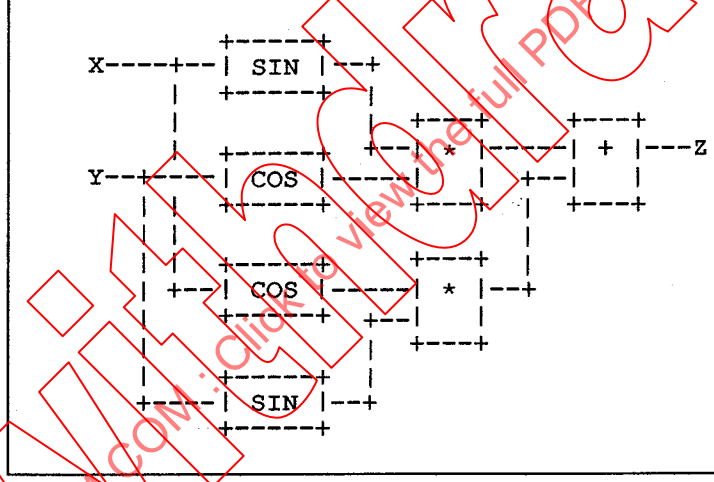


Figure 4 – Examples of function usage
a) Structured Text (ST) language; see subclause 3.3
b) Function Block Diagram (FBD) language; see subclause 4.3

Functions shall contain no internal state information, i.e., invocation of a function with the same arguments (input parameters) shall always yield the same value (output).

Any function type which has already been declared can be used in the declaration of another program organization unit, as shown in figure 3.

2.5.1.1 *Représentation*

Les fonctions et leur lancement peuvent être représentés soit graphiquement, soit littéralement.

Dans les langages graphiques définis à l'article 4 de la présente partie, les fonctions doivent être représentées en tant que blocs graphiques, conformément aux règles suivantes:

1) La forme du bloc doit être rectangulaire ou carrée.

2) La taille et les proportions du bloc peuvent varier en fonction du nombre d'entrées, ainsi que du nombre d'informations à afficher.

3) Le sens du traitement dans le bloc, doit être de gauche à droite (paramètres d'entrée à gauche et paramètre de sortie à droite).

4) Le nom ou le symbole de la fonction, tel que spécifié ci-dessous, doit se trouver à l'intérieur du bloc.

5) Il est nécessaire de prévoir des noms formels pour les paramètres d'entrée; ces noms doivent figurer à la partie intérieure gauche du bloc lorsque le bloc représente:

   – une des fonctions normalisées données au 2.5.1.5, lorsque la forme graphique donnée comprend les noms formels des paramètres, ou

   – une fonction supplémentaire quelconque, déclarée comme spécifié au 2.5.1.3.

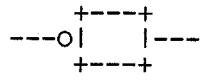
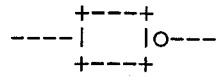
6) Dans la mesure où le nom de la fonction sert à l'affectation de sa valeur de sortie, comme cela est spécifié en 2.5.1.3, il n'est pas nécessaire qu'un nom de paramètre de sortie figure à la partie droite du bloc.

7) Les liaisons réelles entre paramètres doivent être indiquées par des lignes de circulation des signaux.

8) L'inversion des signes booléens doit être indiquée en plaçant un cercle ouvert juste à l'extérieur de l'intersection des lignes d'entrée et de sortie avec le bloc. Dans le jeu de caractères ISO/IEC 646, il doit être représenté par la lettre "O" majuscule, comme l'illustre le tableau 19.

9) La sortie d'une fonction représentée graphiquement doit être représentée par une ligne unique, située à droite du bloc, même s'il est possible que la sortie soit une variable à plusieurs éléments.

Tableau 19 – Inversion graphique de signes booléens

N°	Caractéristiques	Représentation
1	Entrée inversée	
2	Sortie inversée	
NOTE - Si l'une des deux caractéristiques mentionnées ci-dessus est acceptée pour des fonctions, elle doit également être acceptée pour des blocs fonctionnels tels que définis en 2.5.2, et réciproquement.		

2.5.1.1 *Representation*

Functions and their invocation can be represented either graphically or textually.

In the graphic languages defined in clause 4 of this part, functions shall be represented as graphic blocks according to the following rules:

1) The form of the block shall be rectangular or square.

2) The size and proportions of the block may vary depending on the number of inputs and other information to be displayed.

3) The direction of processing through the block shall be from left to right (input parameters on the left and output parameter on the right).

4) The function name or symbol, as specified below, shall be located inside the block.

5) Provision shall be made for formal input parameter names appearing at the inside left of the block when the block represents:

- one of the standard functions defined in subclause 2.5.1.5, when the given graphical form includes the formal parameter names; or
- any additional function declared as specified in subclause 2.5.1.3.

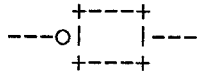
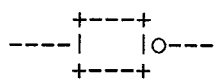
6) Since the name of the function is used for the assignment of its output value as specified in 2.5.1.3, no formal output parameter name shall be shown at the right side of the block.

7) Actual parameter connections shall be shown by signal flow lines.

8) Negation of Boolean signals shall be shown by placing an open circle just outside of the input or output line intersection with the block. In the ISO/IEC 646 character set, this shall be represented by the upper case alphabetic "O", as shown in table 19.

9) The output of a graphically represented function shall be represented by a single line at the right side of the block, even though the output may be a multi-element variable.

Table 19 – Graphical negation of Boolean signals

No.	Feature	Representation
1	Negated input	
2	Negated output	
NOTE - If either of these features is supported for functions, it shall also be supported for function blocks as defined in 2.5.2, and vice versa.		

Comme l'illustre la figure 5, lorsqu'un nom de paramètre formel est présent dans la définition d'une fonction standard, en 2.5.1.5, le nom de paramètre formel doit être également utilisé dans le lancement littéral de la fonction. Dans le dernier cas, les noms de paramètres formels et les valeurs réelles associées peuvent être donnés dans n'importe quel ordre.

La représentation de fonctions en langages littéraux doit être conforme aux spécifications de l'article 3 de la présente partie.

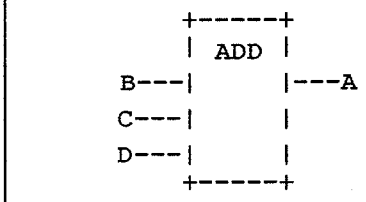
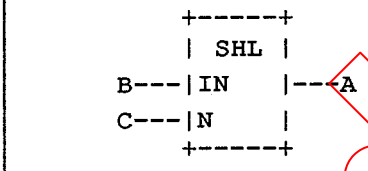
Exemple	Explication
	Utilisation graphique de la fonction "ADD" (Voir 2.5.1.5.2) (FBD langage; voir 4.3) (Aucun nom de paramètre formel)
`A := ADD(B,C,D);`	Utilisation littérale de la fonction "ADD" (Langage ST; voir 3.3)
	Utilisation graphique de la fonction "SHL" (Voir 2.5.1.5.3) (Langage FBD; voir 4.3) (Noms de paramètres formels)
`A := SHL(IN := B,N := C);`	Utilisation littérale de la fonction "SHL" (Langage ST; voir 3.3)

Figure 5 – Utilisation de noms de paramètres formels

2.5.1.2 *Commande d'exécution*

Comme l'illustre le tableau 20, une entrée "EN" (Enable) et une sortie "ENO" (Enable Out) booléennes supplémentaires doivent être utilisées avec les fonctions dans le langage LD défini en 4.2, et leur utilisation doit être également possible dans le langage FBD défini dans la présente partie. On considère que ces variables sont disponibles dans chaque fonction selon les déclarations implicites

VAR_INPUT EN : BOOL := 1 ; END_VAR

VAR_OUTPUT ENO : BOOL ; END_VAR

Lorsque ces variables sont utilisées, l'exécution des opérations définies par la fonction doit être commandée conformément aux règles suivantes:

1) Si la valeur de EN est FALSE (0) lorsque la fonction est lancée, les opérations définies par le corps de la fonction ne doivent pas être exécutées et la valeur de "ENO" doit être remise à FALSE (0) par le système d'automate programmable.

2) Sinon, la valeur de ENO doit être mise à TRUE (1) par le système d'automate programmable, et les opérations définies par le corps de la fonction doivent être exécutées. Ces opérations peuvent inclure l'affectation d'une valeur booléenne à ENO.

As shown in figure 5, where a formal parameter name is present in the definition of a standard function in 2.5.1.5, the formal parameter name shall also be used in the textual invocation of the function. In the latter case, the formal parameter names and associated actual values can be given in any order.

The representation of functions in textual languages shall be as specified in clause 3 of this part.

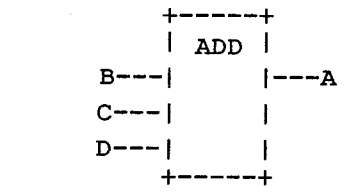
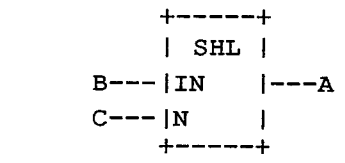
Example	Explanation
	Graphical use of "ADD" function (See 2.5.1.5.2) (FBD language; see 4.3) (No formal parameter names)
`A := ADD(B,C,D);`	Textual use of "ADD" function (ST language; see 3.3)
	Graphical use of "SHL" function (See 2.5.1.5.3) (FBD language; see 4.3) (Formal parameter names)
`A := SHL(IN := B,N := C);`	Textual use of "SHL" function (ST language; see 3.3)

Figure 5 – Use of formal parameter names

2.5.1.2 *Execution control*

As shown in table 20, an additional Boolean "EN" (Enable) input and "ENO" (Enable Out) output shall be used with functions in the LD language defined in 4.2, and their use shall also be possible in the FBD language defined in this part. These variables are considered to be available in every function according to the implicit declarations

```
VAR_INPUT EN : BOOL := 1 ; END_VAR
VAR_OUTPUT ENO : BOOL ; END_VAR
```

When these variables are used, the execution of the operations defined by the function shall be controlled according to the following rules:

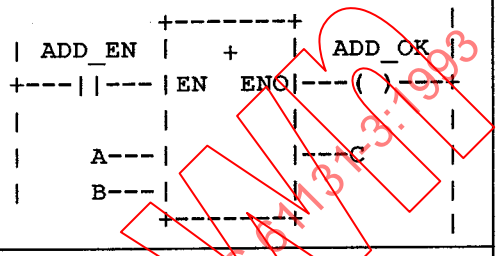
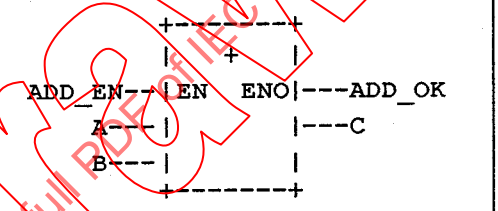
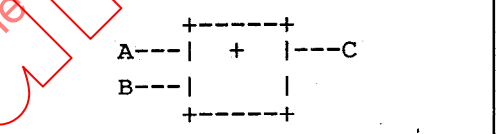
1) If the value of EN is FALSE (0) when the function is invoked, the operations defined by the function body shall not be executed and the value of "ENO" shall be reset to FALSE (0) by the programmable controller system.

2) Otherwise, the value of ENO shall be set to TRUE (1) by the programmable controller system, and the operations defined by the function body shall be executed. These operations can include the assignment of a Boolean value to ENO.

3) Si l'une des erreurs définies à l'annexe E se produit pendant l'exécution de l'une des fonctions standards définies en 2.5.1.5, la sortie ENO de cette fonction doit être remise à FALSE (0) par le système d'automate programmable.

NOTE - L'emploi de la sortie ENO est une exception autorisée à la règle que l'exécution d'une fonction donne exactement une sortie.

Tableau 20 – Utilisation d'une entrée "EN" et d'une sortie "ENO"

N°	Caractéristique	Exemple
1	Utilisation de "EN" et "ENO" Requis pour le langage LD (langage à contacts) (voir 4.2)	
2	Utilisation de "EN" et "ENO" Facultatif pour le langage FBD (schéma en blocs fonctionnels) (voir 4.3)	
3	FBD sans "EN" ni "ENO"	

2.5.1.3 *Déclaration*

Une fonction doit être déclarée littéralement ou graphiquement.

Comme l'illustre la figure 6, la déclaration littérale d'une fonction doit se composer des éléments suivants:

1) le mot clé FUNCTION, suivi d'un identificateur spécifiant le nom de la fonction en cours de déclaration, le symbole deux-points (:), et le type de donnée de la valeur à renvoyer par la fonction;

2) une construction VAR_INPUT...END_VAR, telle que définie en 2.4.2, spécifiant les noms et les types des paramètres d'entrée de la fonction;

3) une construction VAR...END_VAR, si nécessaire, spécifiant les noms et les types des variables internes de la fonction;

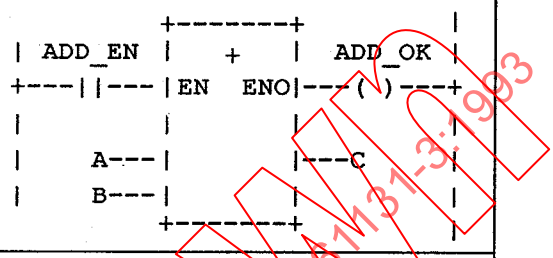
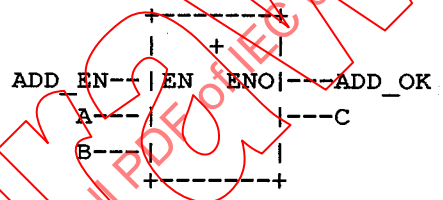
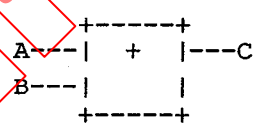
4) un *corps de fonction,* écrit dans l'un des langages définis dans la présente partie, ou un autre langage de programmation, défini en 1.2.3, qui spécifie les opérations à effectuer sur le(s) paramètre(s) d'entrée, afin d'affecter une ou plusieurs valeurs à une variable qui porte le même nom que la fonction et qui représente la valeur que la fonction doit renvoyer;

5) le mot clé de fin END_FUNCTION.

3) If one of the errors defined in annex E occurs during the execution of one of the standard functions defined in 2.5.1.5, the ENO output of that function shall be reset to FALSE (0) by the programmable controller system.

NOTE - The use of the ENO output is an allowable exception to the rule that the execution of a function yields exactly one output.

Table 20 – Use of EN input and ENO output

No.	Feature	Example
1	Use of "EN" and "ENO" Required for LD (Ladder Diagram) language (see 4.2)	
2	Use of "EN" and "ENO" Optional for FBD (Function Block Diagram) language (see 4.3)	
3	FBD without "EN" and "ENO"	

2.5.1.3 *Declaration*

A function shall be declared textually or graphically.

As illustrated in figure 6, the textual declaration of a function shall consist of the following elements:

1) The keyword FUNCTION, followed by an identifier specifying the name of the function being declared, a colon (:), and the data type of the value to be returned by the function;

2) A VAR_INPUT...END_VAR construct as defined in 2.4.2, specifying the names and types of the function's input parameters;

3) A VAR...END_VAR construct, if required, specifying the names and types of the function's internal variables;

4) *A function body*, written in one of the languages defined in this part, or another programming language as defined in 1.4.3, which specifies the operations to be performed upon the input parameter(s) in order to assign one or more values to a variable with the same name as the function, which represents the value to be returned by the function;

5) The terminating keyword END_FUNCTION.

Comme l'illustre la figure 6, la déclaration graphique d'une fonction doit se composer des éléments suivants:

1) les mots clés de mise entre parenthèses FUNCTION...END_FUNCTION ou un équivalent graphique;

2) une spécification graphique du nom de la fonction, ainsi que des noms et des types des entrées et de la sortie de la fonction;

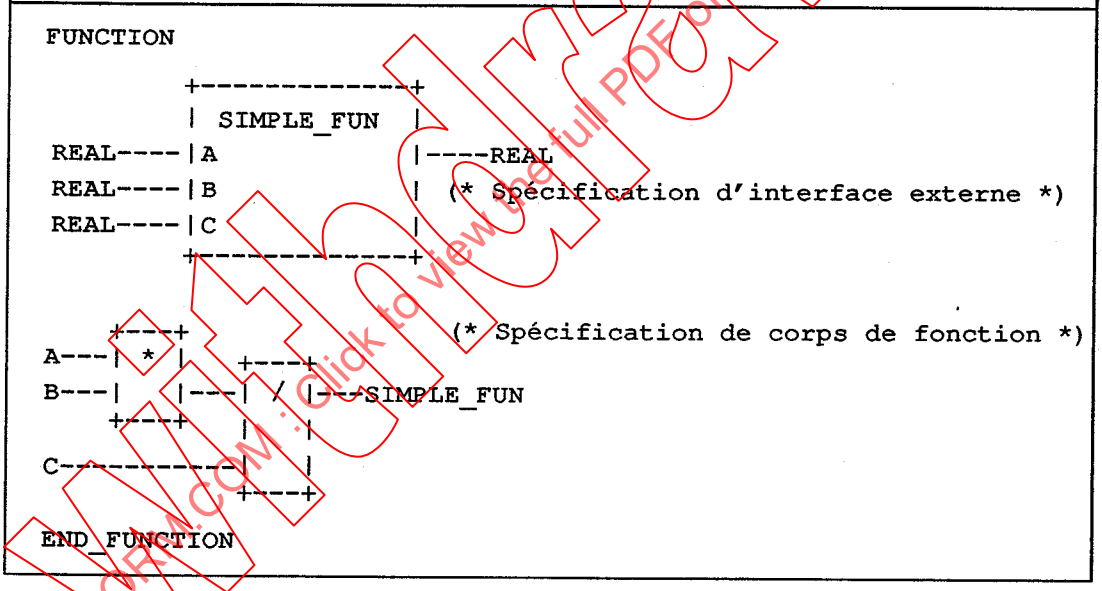
3) une spécification des noms et des types des variables internes utilisées dans la fonction, par exemple en utilisant la construction VAR...END_VAR;

4) un corps de fonction tel que défini plus haut.

a)

```
FUNCTION SIMPLE_FUN : REAL
  VAR_INPUT
    A,B : REAL;          (* Spécification d'interface externe *)
      C : REAL := 1.0;
  END_VAR

  SIMPLE_FUN := A*B/C;  (* Spécification de corps de fonction *)

  END_FUNCTION
```

b)



NOTE – Dans l'exemple a), la variable d'entrée C est affectée d'une valeur 1.0, comme cela est spécifié en 2.4.3.2, afin d'éviter une erreur du type "division par zéro", si l'entrée n'est pas spécifiée lorsque la fonction est lancée; par exemple, si une entrée graphique de la fonction est laissée sans liaison.

Figure 6 – Exemples de déclarations de fonctions
a) Déclaration littérale en langage ST (voir 3.3)
b) Déclaration graphique en langage FBD (voir 4.3)

2.5.1.4 *Identification, surcharge et conversion de types*

On dit qu'une fonction ou une opération est *surchargée* lorsqu'elle peut traiter des éléments de données d'entrée de divers types, au sein d'un indicateur de type générique tel que défini en 2.3.2. Par exemple, une fonction d'addition surchargée sur le type générique ANY_NUM peut traiter des données des types LREAL, REAL, DINT, INT, et SINT.

As illustrated in figure 6, the graphic declaration of a function shall consist of the following elements:

1) The bracketing keywords FUNCTION...END_FUNCTION or a graphical equivalent;

2) A graphic specification of the function name and the names and types of the function's inputs and output;

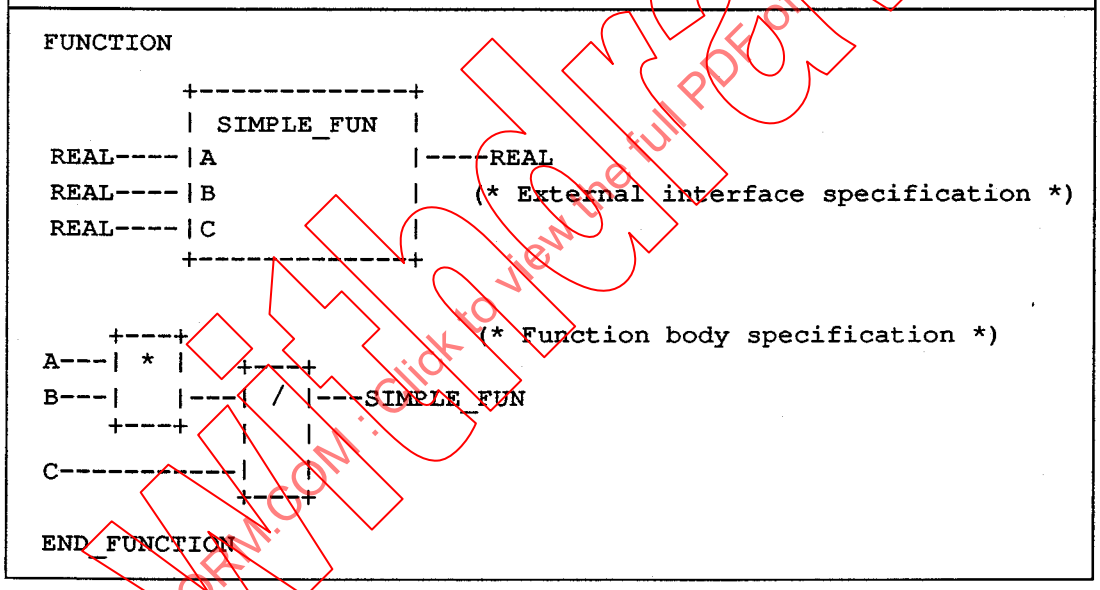
3) A specification of the names and types of the internal variables used in the function, e.g., using the VAR...END_VAR construct;

4) A function body as defined above.

a)

```
FUNCTION SIMPLE_FUN : REAL
  VAR_INPUT
    A,B : REAL;          (* External interface specification *)
      C : REAL := 1.0;
  END_VAR

  SIMPLE_FUN := A*B/C;  (* Function body specification *)

  END_FUNCTION
```

b)



NOTE - In example a), the input variable C is given a default value of 1.0, as specified in 2.4.3.2, to avoid a "division by zero" error if the input is not specified when the function is invoked, for example, if a graphical input to the function is left unconnected.

Figure 6 – Examples of function declarations
a) Textual declaration in ST language (see 3.3)
b) Graphical declaration in FBD language (see 4.3)

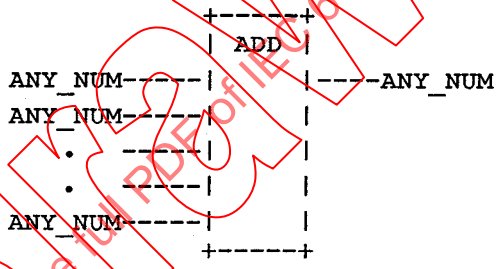
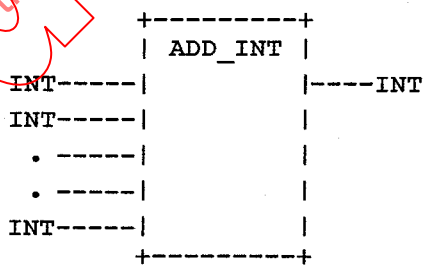
2.5.1.4 *Typing, overloading, and type conversion*

A function or operation is said to be *overloaded* when it can operate on input data elements of various types within a generic type designator as defined in 2.3.2. For instance, an overloaded addition function on generic type ANY_NUM can operate on data of types LREAL, REAL, DINT, INT, and SINT.

Lorsqu'un système d'automate programmable accepte une opération ou une fonction surchargée, cette opération ou fonction doit s'appliquer à tous les types de données du type générique considéré qui sont acceptés par ce système. Par exemple, si un système d'automate programmable accepte la fonction d'addition surchargée ADD et les types de données SINT, INT et REAL, le système doit alors accepter la fonction ADD sur des entrées de type SINT, INT et REAL.

Lorsqu'une fonction, qui représente normalement un opérateur surchargé, doit être *identifiée,* c'est-à-dire lorsque les types de ses entrées et de ses sorties se limitent à un sous-type particulier, ceci doit être effectué en ajoutant un caractère de "soulignement" suivi du type requis, comme l'illustre le tableau 21.

Tableau 21 - Fonctions identifiées et surchargées

N°	Caractéristique	Exemple
1	Fonctions surchargées	
2	Fonctions identifiées	

NOTES

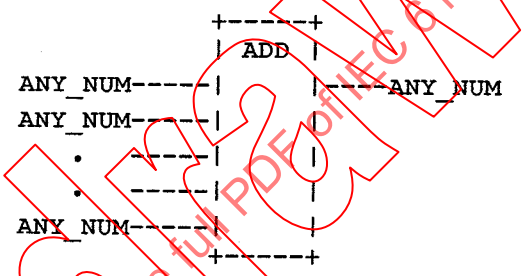
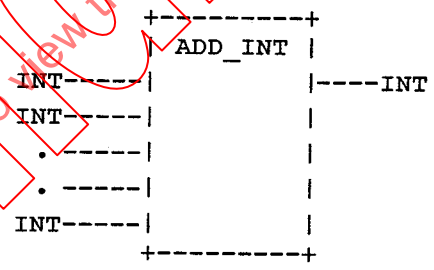
1 Si la caractéristique n° 2 est acceptée, le fabricant doit fournir un tableau indiquant les fonctions surchargées et celles qui sont identifiées dans l'application concernée.

2 Les fonctions définies par l'utilisateur ne peuvent être surchargées.

When a programmable controller system supports an overloaded operation or function, this operation or function shall apply to all data types of the given generic type which are supported by that system. For example, if a programmable controller system supports the overloaded function ADD and the data types SINT, INT, and REAL, then the system shall support the ADD function on inputs of type SINT, INT, and REAL.

When a function which normally represents an overloaded operator is to be typed, i.e., the types of its inputs and outputs restricted to a particular subtype, this shall be done by appending an "underline" character followed by the required type, as shown in table 21.

Table 21 – Typed and overloaded functions

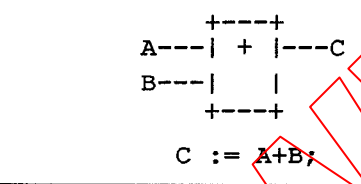
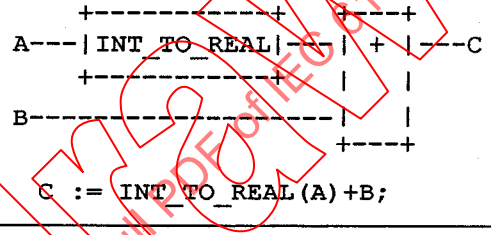
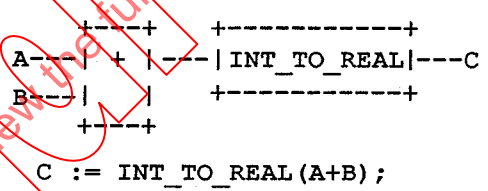
No.	Feature	Example
1	Overloaded functions	
2	Typed functions	

NOTES

1 If feature 2 is supported, the manufacturer shall provide a table of which functions are overloaded and which are typed in the implementation.

2 User-defined functions cannot be overloaded.

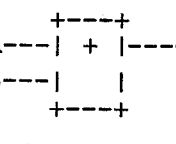
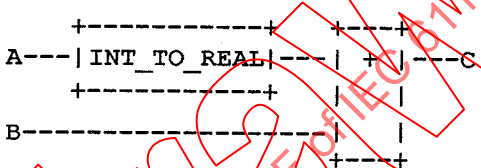
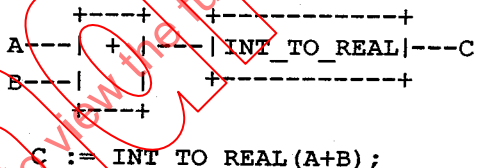
Lorsque tous les paramètres d'entrée formels, se rapportant à une fonction standard définie en 2.5.1.5, sont du même type générique, tous les paramètres réels doivent alors être du même type. Si nécessaire, il est possible d'utiliser les fonctions de conversion de types, indiquées en 2.5.1.5.1, pour satisfaire à cette exigence. La valeur de sortie de la fonction doit alors avoir le même type que les entrées réelles sauf comme noté au tableau 22. Les figures 7 et 8 illustrent des exemples d'application de cette règle.

Déclaration de type (Langage ST - voir 3.3)	Application (Langage FBD - voir 4.3) (Langage ST - voir 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	
VAR A : INT ; B : REAL ; C : REAL ; END_VAR	
VAR A : INT ; B : INT ; C : REAL ; END_VAR	

NOTE - La conversion de type n'est pas requise dans le premier cas ci-dessus.

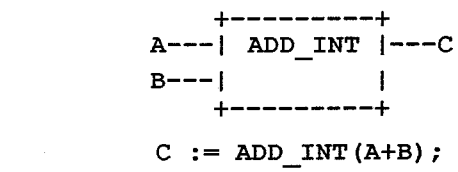
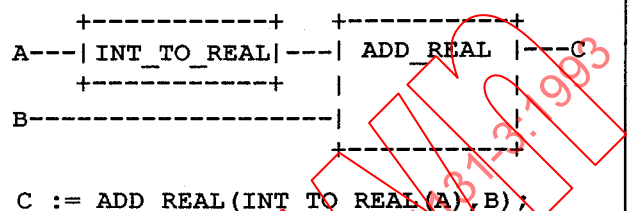
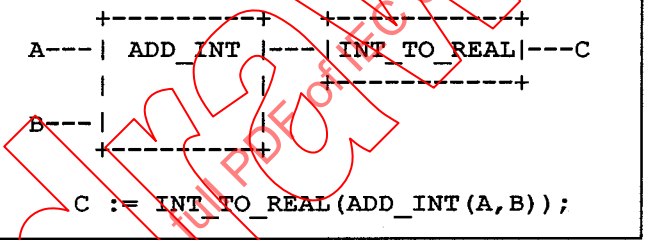
Figure 7 – Exemples de conversion de type explicite, avec des fonctions surchargées

When all the formal input parameters to a standard function defined in 2.5.1.5 are of the same generic type then all the actual parameters shall be of the same type. If necessary, the type conversion functions defined in 2.5.1.5.1 can be used to meet this requirement. The output value of the function shall then have the same type as the actual inputs, except as noted in table 22. Examples of the application of this rule are given in figures 7 and 8.

Type declaration (ST language - see 3.3)	Usage (FBD language - see 4.3) (ST language - see 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	 C := A+B;
VAR A : INT ; B : REAL ; C : REAL ; END_VAR	 C := INT_TO_REAL(A)+B;
VAR A : INT ; B : INT ; C : REAL ; END_VAR	 C := INT_TO_REAL(A+B);

NOTE - Type conversion is not required in the first example shown above.

Figure 7 – Examples of explicit type conversion with overloaded functions

Déclaration de type (Langage ST - voir 3.3)	Application (Langage FBD - voir 4.3) (Langage ST - voir 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	
VAR A : INT ; B : REAL ; C : REAL ; END_VAR	
VAR A : INT ; B : INT ; C : REAL ; END_VAR	

NOTE - La conversion de type n'est pas requise dans le premier cas ci-dessus.

Figure 8 – Exemples de conversion de type explicité, avec des fonctions identifiées

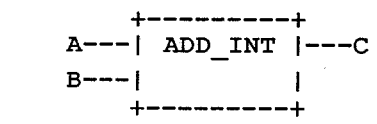
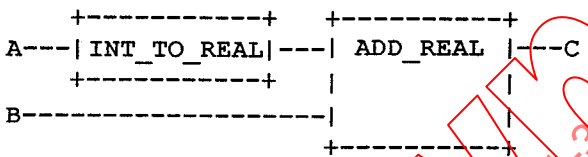
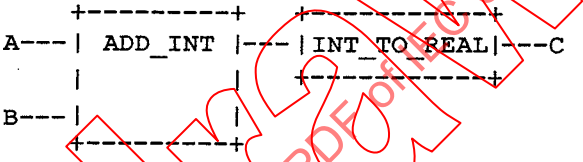
2.5.1.5 *Fonctions standards*

Le présent paragraphe donne des définitions de fonctions communes à tous les langages de programmation d'automates programmables. Lorsque des représentations graphiques de fonctions standards sont illustrées dans le présent paragraphe, des déclarations littérales équivalentes peuvent être écrites conformément aux prescriptions de 2.5.1.3.

Il est admis qu'une fonction standard, spécifiée comme *extensible* dans le présent paragraphe, ait un nombre variable d'entrées; on doit considérer que cette fonction applique, à tour de rôle, l'opération indiquée à chaque entrée; par exemple une addition extensible doit donner, en tant que valeur de sortie, la somme de toutes ses entrées. Le nombre maximal d'entrées d'une fonction extensible est un paramètre propre à l'application concernée.

2.5.1.5.1 *Fonctions de conversion de types*

Comme l'illustre le tableau 22, les fonctions de conversion de types doivent se présenter sous la forme *_TO_**, où "*" est le type de la variable d'entrée IN et "**" est le type de la variable de sortie OUT. Exemple: INT_TO_REAL.

Type declaration (ST language - see 3.3)	Usage (FBD langage - see 4.3) (ST language - see 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	 C := ADD_INT(A+B) ;
VAR A : INT ; B : REAL ; C : REAL ; END_VAR	 C := ADD_REAL(INT_TO_REAL(A),B) ;
VAR A : INT ; B : INT ; C : REAL ; END_VAR	 C := INT_TO_REAL(ADD_INT(A,B)) ;

NOTE - Type conversion is not required in the first example shown above.

Figure 8 – Examples of explicit type conversion with typed functions

2.5.1.5 *Standard functions*

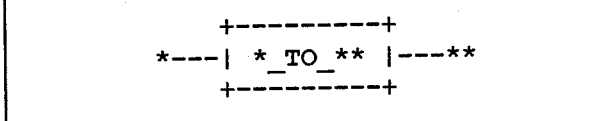
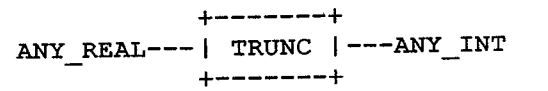
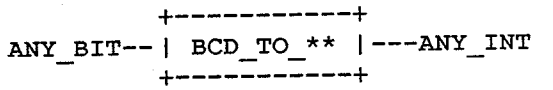
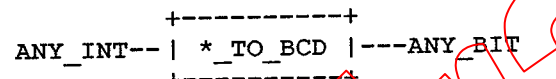
Definitions of functions common to all programmable controller programming languages are given in this subclause. Where graphical representations of standard functions are shown in this subclause, equivalent textual declarations may be written as specified in 2.5.1.3.

A standard function specified in this subclause to be *extensible* is allowed to have a variable number of inputs, and shall be considered as applying the indicated operation to each input in turn, e.g., extensible addition shall give as its output the sum of all its inputs. The maximum number of inputs of an extensible function is an implementation-dependent parameter.

2.5.1.5.1 *Type conversion functions*

As shown in table 22, type conversion functions shall have the form *_TO_**, where "*" is the type of the input variable IN, and "***" the type of the output variable OUT, e.g., INT_TO_REAL.

Tableau 22 – Caractéristiques des fonctions de conversion de types

N°	Forme graphique	Exemple d'application	Notes
1	 (*) Type de donnée d'entrée; exemple: INT (**) Type de donnée de sortie; exemple: REAL (*_TO_**) Nom de fonction; exemple: INT_TO_REAL	A := INT_TO_REAL(B) ;	1 2 5
2		A := TRUNC(B) ;	3
3		A := BCD_TO_INT(B) ;	4
4		A := INT_TO_BCD(B) ;	4

NOTES

1 Un énoncé de conformité à la caractéristique n° 1 de ce tableau doit inclure une liste des conversions de types spécifiques acceptées, ainsi qu'un énoncé des effets de l'exécution de chaque conversion.

2 La conversion du type REAL ou LREAL en type SINT, INT, DINT ou LINT, doit être arrondie selon les conventions de la CEI 559, par exemple:

REAL_TO_INT(1.6) est équivalent à 2
REAL_TO_INT(-1.6) est équivalent à -2
REAL_TO_INT(1.5) est équivalent à 2
REAL_TO_INT(-1.5) est équivalent à -2
REAL_TO_INT(1.4) est équivalent à 1
REAL_TO_INT(-1.4) est équivalent à -1
REAL_TO_INT(2.5) est équivalent à 2
REAL_TO_INT(-2.5) est équivalent à -2

3 La fonction TRUNC doit être utilisée pour la troncation vers zéro d'un type REAL ou LREAL, donnant l'un des types d'entiers, par exemple:

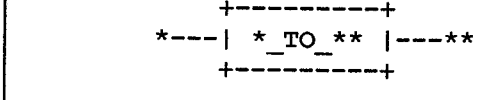
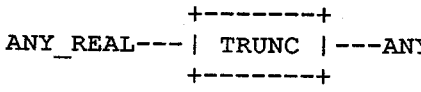
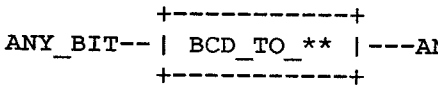
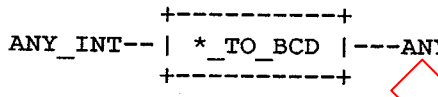
TRUNC(1.6) est équivalent à 1
TRUNC(-1.6) est équivalent à -1
TRUNC(1.4) est équivalent à 1
TRUNC(-1.4) est équivalent à -1

4 Les fonctions de conversion *_TO_BCD et BCD_TO_** sont définies pour effectuer des conversions entre des variables de type BYTE, WORD, DWORD et LWORD d'une part et des variables de type SINT, INT et DINT (représentées par "*") d'autre part, lorsque les variables à cordons de bits correspondantes contiennent des données codées en format BCD. Par exemple, la valeur de INT_TO_BCD(25) devrait être égale à 2#0010_0101, et la valeur de BCD_TO_INT (2#0011_0110_1001) devrait être égale à 369.

5 Lorsqu'une entrée ou une sortie d'une fonction de conversion de types est du type STRING, les données du cordon de caractères doivent être conformes à la représentation externe des données correspondantes, comme cela est spécifié en 2.2, dans le jeu de caractères ISO/IEC 646.

6 Des exemples d'application sont donnés en langage ST défini en 3.3

Table 22 – Type conversion function features

No.	Graphical form	Usage example	Notes
1	 (*) Input data type, e.g., INT (**) Output data type, e.g., REAL (*_TO_**) Function name, e.g., INT_TO_REAL	A := INT_TO_REAL(B) ;	1 2 5
2		A := TRUNC(B) ;	3
3		A := BCD_TO_INT(B) ;	4
4		A := INT_TO_BCD(B) ;	4

NOTES

1 A statement of conformance to feature 1 of this table shall include a list of the specific type conversions supported, and a statement of the effects of performing each conversion.

2 Conversion from type REAL or LREAL to SINT, INT, DINT, or LINT shall round according to the conventions of IEC 559, e.g.:

REAL_TO_INT(1.6) is equivalent to 2
REAL_TO_INT(-1.6) is equivalent to -2
REAL_TO_INT(1.5) is equivalent to 2
REAL_TO_INT(-1.5) is equivalent to -2
REAL_TO_INT(1.4) is equivalent to 1
REAL_TO_INT(-1.4) is equivalent to -1
REAL_TO_INT(2.5) is equivalent to 2
REAL_TO_INT(-2.5) is equivalent to -2

3 The function TRUNC shall be used for truncation toward zero of a REAL or LREAL, yielding one of the integer types, for instance,

TRUNC(1.6) is equivalent to 1
TRUNC(-1.6) is equivalent to -1
TRUNC(1.4) is equivalent to 1
TRUNC(-1.4) is equivalent to -1

4 The conversion functions *_TO_BCD and BCD_TO_** are defined to perform conversions between variables of type BYTE, WORD, DWORD, and LWORD and variables of type SINT, INT, and DINT (represented by "*"), when the corresponding bit-string variables contain data encoded in BCD format. For example, the value of INT_TO_BCD(25) would be 2#0010_0101, and the value of BCD_TO_INT (2#0011_0110_1001) would be 369.

5 When an input or output of a type conversion function is of type STRING, the character string data shall conform to the external representation of the corresponding data, as specified in 2.2, in the ISO/IEC 646 character set.

6 Usage examples are given in the ST language defined in 3.3.

2.5.1.5.2 *Fonctions numériques*

La représentation graphique standard, les noms de fonctions, les types de variables d'entrée et de sortie, ainsi que les descriptions fonctionnelles des fonctions à une seule variable numérique, doivent être tels que définis au tableau 23. Ces fonctions doivent être surchargées sur les types génériques définis, et peuvent être saisies conformément aux spécifications de 2.5.1.4. Pour ces fonctions, les types de l'entrée et de la sortie doivent être identiques.

La représentation graphique standard, les noms et les symboles des fonctions, ainsi que les descriptions des fonctions arithmétiques à au moins deux variables, doivent être tels qu'indiqués au tableau 24. Ces fonctions doivent être surchargées sur tous les types numériques, et peuvent être saisies conformément aux spécifications de 2.5.1.4.

Tableau 23 – Fonctions standards à une seule variable numérique

Forme graphique			Exemple d'application
+---------+ *---| ** |---* +---------+ (*) – Type entrée/sortie (E/S) (**) – Nom de fonction			A := SIN(B) ; (Langage ST – voir 3.3)
N°	Nom de fonction	Type d'E/S	Description
	Fonctions générales		
1	ABS	ANY_NUM	Valeur absolue
2	SQRT	ANY_REAL	Racine carrée
	Fonctions logarithmiques		
3	LN	ANY_REAL	Logarithme naturel
4	LOG	ANY_REAL	Logarithme en base 10
5	EXP	ANY_REAL	Exponentielle naturelle
	Fonctions trigonométriques		
6	SIN	ANY_REAL	Sinus d'entrée en radians
7	COS	ANY_REAL	Cosinus d'entrée en radians
8	TAN	ANY_REAL	Tangente d'entrée en radians
9	ASIN	ANY_REAL	Arc sinus principal
10	ACOS	ANY_REAL	Arc cosinus principal
11	ATAN	ANY_REAL	Arc tangente principal

2.5.1.5.2 *Numerical functions*

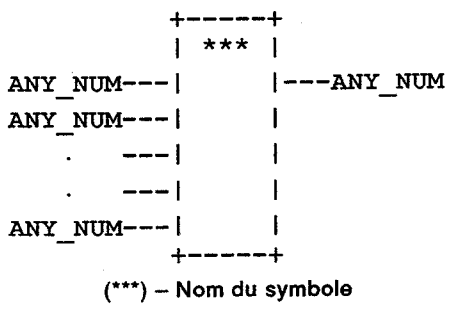
The standard graphical representation, function names, input and output variable types, and function descriptions of functions of a single numeric variable shall be as defined in table 23. These functions shall be overloaded on the defined generic types, and can be typed as defined in 2.5.1.4. For these functions, the types of the input and output shall be the same.

The standard graphical representation, function names and symbols, and descriptions of arithmetic functions of two or more variables shall be as shown in table 24. These functions shall be overloaded on all numeric types, and can be typed as defined in 2.5.1.4.

Table 23 – Standard functions of one numeric variable

Graphical form			Usage example
+---------+ *---| ** |---* +---------+ (*) – Input/Output (I/O) type (**) – Function name			A := SIN(B) ; (ST language – see 3.3)
No.	Function name	I/O type	Description
	General functions		
1	ABS	ANY_NUM	Absolute value
2	SQRT	ANY_REAL	Square root
	Logarithmic functions		
3	LN	ANY_REAL	Natural logarithm
4	LOG	ANY_REAL	Logarithm base 10
5	EXP	ANY_REAL	Natural exponential
	Trigonometric functons		
6	SIN	ANY_REAL	Sine of input in radians
7	COS	ANY_REAL	Cosine in radians
8	TAN	ANY_REAL	Tangent in radians
9	ASIN	ANY_REAL	Principal arc sine
10	ACOS	ANY_REAL	Principal arc cosine
11	ATAN	ANY_REAL	Principal arc tangent

Tableau 24 – Fonctions arithmétiques extensibles

Forme graphique	Exemple d'application
 (***) – Nom du symbole	A := ADD(B,C,D) ; ou A := B+C+D ;

N°	Nom	Symbole (note 1)	Description (notes 2 et 8)	
	Fonctions arithmétiques extensibles			
12	ADD	+	OUT := IN1 + IN2 +...+ INn	
13	MUL	*	OUT := IN1 * IN2 *...* INn	
	Fonctions arithmétiques inextensibles			
14	SUB	–	OUT := IN1 – IN2	
15	DIV	/	OUT := IN1 / IN2	(note 5)
16	MOD		OUT := IN1 modulo IN2	(note 3)
17	EXPT	**	Exponentiation: OUT := $IN1^{IN2}$	(note 4)
18	MOVE	:=	OUT := IN	(note 9)

NOTES

1 Ces symboles conviennent à l'utilisation en tant qu'opérateurs dans des langages littéraux, comme l'indiquent les tableaux 52 et 55.

2 Les notations IN1, IN2, ..., INn se rapportent aux entrées dans un ordre descendant (du haut vers le bas); OUT se rapporte à la sortie.

3 En ce qui concerne cette fonction, IN1 et IN2 doivent être du type générique ANY_INT. Le résultat de l'évaluation de cette fonction doit être équivalent à l'exécution des énoncés suivants en langage ST, tel que défini en 3.3:

IF (IN2 = 0) THEN OUT := 0 ; ELSE OUT := IN1 - (IN1/IN2)*IN2 ; END_IF

4 En ce qui concerne cette fonction, IN1 doit être du type ANY_REAL, et IN2 doit être du type ANY_NUM. La sortie doit être du même type que IN1.

5 Le résultat de la division de deux entiers doit être un entier du même type, avec troncation vers zéro; par exemple: 7/3 = 2 et (–7)/3 = –2.

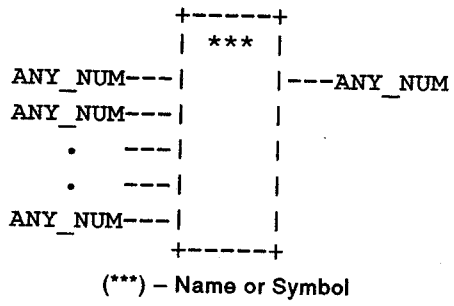
6 Lorsque la représentation nommée d'une fonction est acceptée, cela doit être indiqué par le suffixe "n" dans l'énoncé de conformité. Par exemple: "12n" représente la notation "ADD".

7 Lorsque la représentation symbolique d'une fonction est acceptée, cela doit être indiqué par le suffixe "s" dans l'énoncé de conformité. Par exemple: "12s" représente la notation "+".

8 Des exemples d'application et des descriptions sont donnés en langage ST, tel qu'il est défini au en 3.3.

9 La fonction MOVE a exactement une entrée (IN) du type ANY et une sortie (OUT) de type ANY.

Table 24 – Standard arithmetic functions

Graphical form	Usage example
	A := ADD(B,C,D) ; or A := B+C+D ;

No.	Name	Symbol (note 1)	Description (notes 2 and 8)	
	Extensible arithmetic functions			
12	ADD	+	OUT := IN1 + IN2 +...+ INn	
13	MUL	*	OUT := IN1 * IN2 *...* INn	
	Non-extensible arithmetic functions			
14	SUB	–	OUT := IN1 – IN2	
15	DIV	/	OUT := IN1 / IN2	(note 5)
16	MOD		OUT := IN1 modulo IN2	(note 3)
17	EXPT	**	Exponentiation: OUT := $IN1^{IN2}$	(note 4)
18	MOVE	:=	OUT := IN	(note 9)

NOTES

1 These symbols are suitable for use as operators in textual languages, as shown in tables 52 and 55.

2 The notations IN1, IN2, ..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

3 IN1 and IN2 shall be of generic type ANY_INT for this function. The result of evaluating this function shall be the equivalent of executing the following statements in the ST language as defined in 3.3:

IF (IN2 = 0) THEN OUT := 0 ; ELSE OUT := IN1 - (IN1/IN2)*IN2 ; END_IF

4 IN1 shall be of type ANY_REAL, and IN2 of type ANY_NUM for this function. The output shall be of the same type as IN1.

5 The result of division of integers shall be an integer of the same type with truncation toward zero, for instance, 7/3 = 2 and (-7)/3 = -2.

6 When the named representation of a function is supported, this shall be indicated by the suffix "n" in the compliance statement. For example, "12n" represents the notation "ADD".

7 When the symbolic representation of a function is supported, this shall be indicated by the suffix "s" in the compliance statement. For example, "12s" represents the notation "+".

8 Usage examples and descriptions are given in the ST language defined in 3.3.

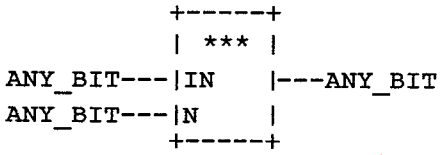
9 The MOVE function has exactly one input (IN) of type ANY and one output (OUT) of type ANY.

2.5.1.5.3 *Fonctions de cordons de bits*

La représentation graphique standard, les noms de fonctions et les descriptions de fonctions de décalage relatives à une variable unique de cordons de bits doivent être tels que définis au tableau 25. Ces fonctions doivent être surchargées sur tous les types de cordons de bits, et peuvent être saisies comme cela est défini en 2.5.1.4.

La représentation graphique standard, les noms et les symboles de fonctions, ainsi que les descriptions de fonctions booléennes au niveau du bit doivent être tels que définis au tableau 26. Ces fonctions doivent être extensibles, sauf pour la fonction NOT, et surchargées sur tous les types de cordons de bits, et peuvent être saisies conformément à 2.5.1.4.

Tableau 25 – Fonctions standards de décalage binaire

Forme graphique	Exemple d'application
 (***) – Nom de fonction	A := SHL(IN := B, N := 5) ; (Langage ST – voir 3.3)

N°	Nom	Description
1	SHL	OUT := IN Décalage à gauche de N bits, remplissage de zéros à droite
2	SHR	OUT := IN Décalage à droite de N bits, remplissage de zéros à gauche
3	ROR	OUT := IN Rotation à droite de N bits, circulaire
4	ROL	OUT := IN Rotation à gauche de N bits, circulaire

NOTE – La notation "OUT" se rapporte à la sortie de la fonction.

2.5.1.5.4 *Fonctions de sélection et de comparaison*

Les fonctions de sélection et de comparaison doivent être surchargées sur tous les types de données. Les représentations graphiques standards, les noms et les descriptions de fonctions de sélection doivent être conformes aux indications du tableau 27.

La représentation graphique standard, les noms et les symboles de fonctions, ainsi que les descriptions de fonctions de comparaison, doivent être tels que définis au tableau 28. Toutes les fonctions de comparaison (sauf la fonction NE) doivent être extensibles.

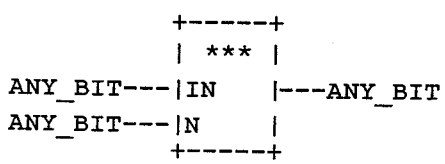
Les comparaisons de données de cordons de bits doivent être effectuées au niveau du bit, depuis le bit le plus significatif jusqu'au bit le moins significatif; les cordons de bits plus courts doivent être considérés comme remplis de zéros à gauche lorsqu'ils sont comparés à des cordons de bits plus longs; c'est-à-dire: la comparaison de variables de cordons de bits doit donner le même résultat que la comparaison de variables d'entiers non signés.

2.5.1.5.3 *Bit-string functions*

The standard graphical representation, function names and descriptions of shift functions for a single bit-string variable shall be as defined in table 25. These functions shall be overloaded on all bit-string types, and can be typed as defined in 2.5.1.4.

The standard graphical representation, function names and symbols, and descriptions of bitwise Boolean functions shall be as defined in table 26. These functions shall be extensible, except for NOT, and overloaded on all bit-string types, and can be typed as defined in 2.5.1.4.

Table 25 – Standard bit-shift functions

Graphical form	Usage example
 (***) – Function Name	A := SHL(IN := B, N := 5) ; (ST language – see 3.3)

No.	Name	Description
1	SHL	OUT := IN left-shifted by N bits, zero-filled on right
2	SHR	OUT := IN right-shifted by N bits, zero-filled on left
3	ROR	OUT := IN right-rotated by N bits, circular
4	ROL	OUT := IN left-rotated by N bits, circular
NOTE – The notation "OUT" refers to the function output.		

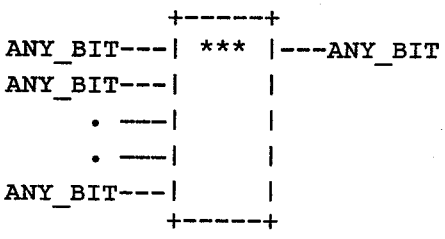
2.5.1.5.4 *Selection and comparison functions*

Selection and comparison functions shall be overloaded on all data types. The standard graphical representations, function names and descriptions of selection functions shall be as shown in table 27.

The standard graphical representation, function names and symbols and descriptions of comparison functions shall be as defined in table 28. All comparison functions (except NE) shall be extensible.

Comparisons of bit-string data shall be made bitwise from the most significant to the least significant bit, and shorter bit strings shall be considered to be filled on the left with zeros when compared to longer bit strings; that is, comparison of bit-string variables shall have the same result as comparison of unsigned integer variables.

Tableau 26 – Fonctions booléennes standards au niveau du bit

Forme graphique	Exemple d'application
 (***) – Nom ou symbole	A := AND(B,C,D) ; ou A := B & C & D ;

N°	Nom	Symbole	Description
5	AND	& (note 1)	OUT := IN1 & IN2 & ... & INn
6	OR	>=1 (note 2)	OUT := IN1 OR IN2 OR ... OR INn
7	XOR	=2k+1 (note 2)	OUT := IN1 XOR IN2 XOR ... XOR INn
8	NOT		OUT := NOT IN1 (note 4)

NOTES

1 Ce symbole convient à l'utilisation en tant qu'opérateur dans des langages littéraux, comme l'illustrent les tableaux 52 et 55.

2 Ce symbole ne convient pas à l'utilisation en tant qu'opérateur dans des langages littéraux.

3 Les notations IN1, IN2, ..., INn se rapportent aux entrées dans un ordre descendant (haut vers le bas); OUT se rapporte à la sortie.

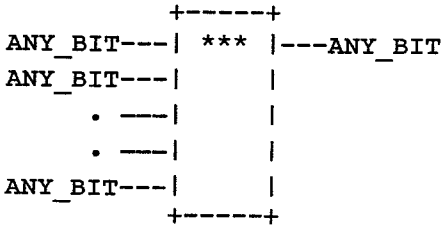
4 L'inversion graphique de signes de type BOOL peut être également effectuée conformément aux indications du tableau 19.

5 Lorsque la représentation nommée d'une fonction est acceptée, cela doit être indiqué par le suffixe "n" dans l'énoncé de conformité. Par exemple: "5n" représente la notation "AND".

6 Lorsque la représentation symbolique d'une fonction est acceptée, cela doit être indiqué par le suffixe "s" dans l'énoncé de conformité. Par exemple: "5s" représente la notation "&".

7 Des exemples d'application et des descriptions sont donnés en langage ST, tel qu'il est défini en 3.3.

Table 26 – Standard bitwise Boolean functions

Graphical form	Usage example
 (***) – Name or symbol	A := AND(B,C,D) ; or A := B & C & D ;

No.	Name	Symbol	Description
5	AND	& (note 1)	OUT := IN1 & IN2 & ... & INn
6	OR	>=1 (note 2)	OUT := IN1 OR IN2 OR ... OR INn
7	XOR	=2k+1 (note 2)	OUT := IN1 XOR IN2 XOR ... XOR INn
8	NOT		OUT := NOT IN1 (note 4)

NOTES

1 This symbol is suitable for use as an operator in textual languages, as shown in tables 52 and 55.

2 This symbol is not suitable for use as an operator in textual languages.

3 The notations IN1, IN2, ..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

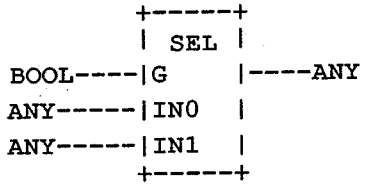
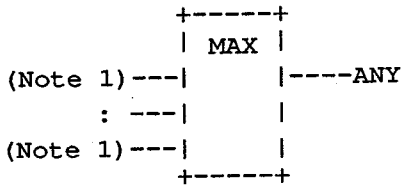
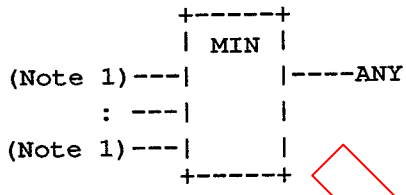
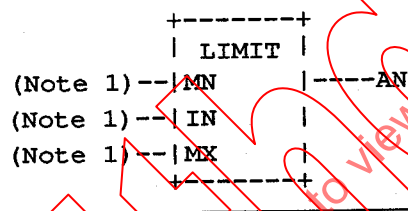
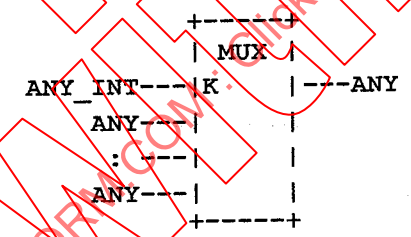
4 Graphic negation of signals of type BOOL can also be accomplished as shown in table 19.

5 When the named representation of a function is supported, this shall be indicated by the suffix "n" in the compliance statement. For example, "5n" represents the notation "AND".

6 When the symbolic representation of a function is supported, this shall be indicated by the suffix "s" in the compliance statement. For example, "5s" represents the notation "&".

7 Usage examples and descriptions are given in the ST language defined in 3.3.

Tableau 27 – Fonctions standards de sélection

N°	Forme graphique	Explication/exemple
1		Sélection binaire: OUT := IN0 si G = 0 OUT := IN1 si G = 1 Exemple: A := SEL(G := 0, IN0 := X, IN1 := 5) ;
2a		Fonction extensible au maximum: OUT := MAX(IN1,IN2,...,INn) Exemple: A := MAX(B,C,D) ;
2b		Fonction extensible au minimum: OUT := MIN(IN1,IN2,...,INn) Exemple: A := MIN(B,C,D) ;
3		Limiteur: OUT := MIN(MAX(IN,MN),MX) Exemple: A := LIMIT(IN := B, MN := 0, MX := 5) ;
4		Multiplexeur extensible: Sélection d'une entrée parmi "N" entrées en fonction de K d'entrée Exemple: A :=MUX(K := 0, IN0 :=B, IN1 := C, IN2 := D) ; devrait avoir le même effet que: A := B ;

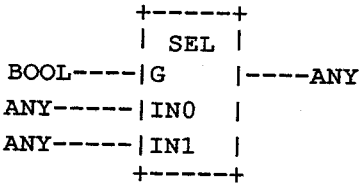
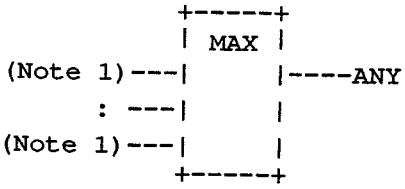
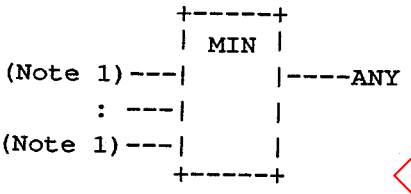
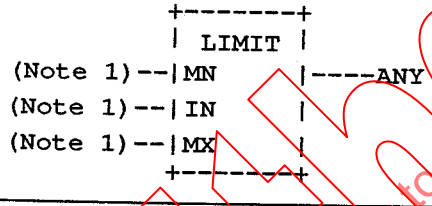
NOTES

1 Ces entrées peuvent être du type ANY_BIT, ANY_NUM, STRING, ANY_DATE, ou TIME. Les règles de conversion de types, définies en 2.5.1.4, doivent être suivies pour ces entrées.

2 Les notations IN1, IN2, ..., INn se rapportent aux entrées dans un ordre descendant (du haut vers le bas); OUT se rapporte à la sortie.

3 Des exemples d'application et des descriptions sont donnés en langage ST, tel qu'il est défini en 3.3.

Table 27 – Standard selection functions

No.	Graphical form	Explanation/example
1		Binary selection: OUT := IN0 if G = 0 OUT := IN1 if G = 1 Example: A := SEL(G := 0, IN0 := X, IN1 := 5) ;
2a		Extensible maximum function: OUT := MAX(IN1,IN2,...,INn) Example: A := MAX(B,C,D) ;
2b		Extensible minimum function: OUT := MIN(IN1,IN2,...,INn) Example: A := MIN(B,C,D) ;
3		Limiter: OUT := MIN(MAX(IN,MN),MX) Example: A := LIMIT(IN := B, MN := 0, MX := 5) ;
4		Extensible multiplexer: Select one of "N" inputs depending on input K Example: A :=MUX(K := 0, IN0 :=B, IN1 := C, IN2 := D) ; would have the same effect as A := B ;

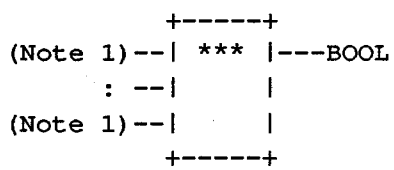
NOTES

1 These inputs can be of type ANY_BIT, ANY_NUM, STRING, ANY_DATE, or TIME. The type conversion rules given in 2.5.1.4 shall be followed for these inputs.

2 The notations IN1, IN2, ..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

3 Usage examples and descriptions are given in the ST language defined in 3.3.

Tableau 28 – Fonctions standards de comparaison

Forme graphique	Exemple d'application
	A := GT(B,C,D) ; ou A := (B>C) & (C>D) ;

N°	Nom	Symbole	Description
5	GT	>	Séquence décroissante: OUT := (IN1>IN2) & (IN2>IN3) & ... & (INn-1 > INn)
6	GE	>=	Séquence monotone: OUT := (IN1>=IN2) & (IN2>=IN3) & ... & (INn-1 >= INn)
7	EQ	=	Egalité: OUT := (IN1=IN2) & (IN2=IN3) & ... & (INn-1 = INn)
8	LE	<=	Séquence monotone: OUT := (IN1<=IN2) & (IN2<=IN3) & ... & (INn-1 <= INn)
9	LT	<	Séquence croissante: OUT := (IN1<IN2) & (IN2<IN3) & ... & (INn-1 < INn)
10	NE	< >	Inégalité (inextensible) OUT := (IN1 <> IN2)

NOTES

1 Ces entrées peuvent être du type ANY_BIT, ANY_NUM, STRING, ANY_DATE, ou TIME. Les règles de conversion de types, définies en 2.5.1.4, doivent être suivies pour ces entrées.

2 Les notations IN1, IN2, ..., INn se rapportent aux entrées dans un ordre descendant (du haut vers le bas); OUT se rapporte à la sortie.

3 Les symboles indiqués dans ce tableau conviennent à l'utilisation en tant qu'opérateurs dans des langages littéraux, comme l'illustrent les tableaux 52 et 55.

4 Lorsque la représentation nommée d'une fonction est acceptée, cela doit être indiqué par le suffixe "n" dans l'énoncé de conformité. Par exemple: "5n" représente la notation "GT".

5 Lorsque la représentation symbolique d'une fonction est acceptée, cela doit être indiqué par le suffixe "s" dans l'énoncé de conformité. Par exemple: "5s" représente la notation ">".

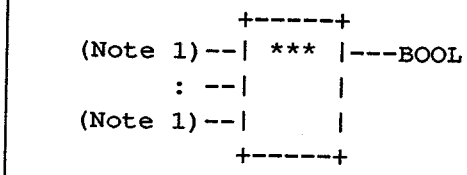
6 Des exemples d'application et des descriptions sont donnés en langage ST, tel qu'il est défini en 3.3.

2.5.1.5.5 *Fonctions de cordons de caractères*

Toutes les fonctions définies en 2.5.1.5.4 doivent être applicables aux cordons de caractères. A des fins de comparaison entre deux cordons de longueur inégale, on doit considérer que le cordon le plus court est étendu à droite jusqu'à la longueur du cordon le plus long, par des caractères ayant la valeur zéro. La comparaison doit s'effectuer de gauche à droite, en se fondant sur la valeur numérique des codes de caractères de la table de codes ISO/IEC 646; par exemple, le cordon de caractères 'Z' doit être supérieur au cordon de caractères 'AZ' et 'AZ' doit être supérieur à 'ABC'.

Les représentations graphiques standards, les noms de fonctions et les descriptions de fonctions supplémentaires de cordons de caractères doivent être conformes aux indications du tableau 29. Aux fins de ces opérations, on doit considérer que les positions des caractères dans le cordon sont numérotées de la façon suivante: 1,2,...,L, en commençant par la position de caractère la plus à gauche, où L est la longueur du cordon.

Table 28 – Standard comparison functions

Graphical form	Usage examples
 (***) – Name or Symbol	A := GT(B,C,D) ; or A := (B>C) & (C>D) ;

No.	Name	Symbol	Description
5	GT	>	Decreasing sequence: OUT := (IN1>IN2) & (IN2>IN3) & ... & (INn-1 > INn)
6	GE	>=	Monotonic sequence: OUT := (IN1>=IN2) & (IN2>=IN3) & ... & (INn-1 >= INn)
7	EQ	=	Equality: OUT := (IN1=IN2) & (IN2=IN3) & ... & (INn-1 = INn)
8	LE	<=	Monotonic sequence: OUT := (IN1<=IN2) & (IN2<=IN3) & ... & (INn-1 <= INn)
9	LT	<	Increasing sequence: OUT := (IN1<IN2) & (IN2<IN3) & ... & (INn-1 < INn)
10	NE	< >	Inequality (non-extensible) OUT := (IN1 <> IN2)

NOTES

1 These inputs can be of type ANY_BIT, ANY_NUM, STRING, ANY_DATE, or TIME. The type conversion rules given in 2.5.1.4 shall be followed for these inputs.

2 The notations IN1, IN2, ..., INn refer to the inputs in top-to-bottom order; OUT refers to the output.

3 All the symbols shown in this table are suitable for use as operators in textual languages, as shown in tables 52 and 55.

4 When the named representation of a function is supported, this shall be indicated by the suffix "n" in the compliance statement. For example, "5n" represents the notation "GT".

5 When the symbolic representation of a function is supported, this shall be indicated by the suffix "s" in the compliance statement. For example, "5s" represents the notation ">".

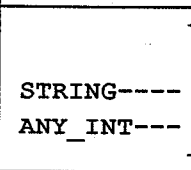
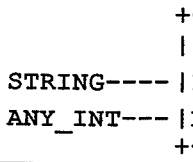
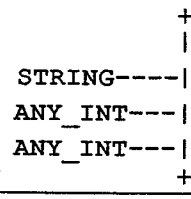
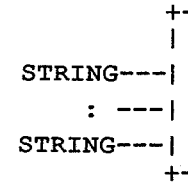
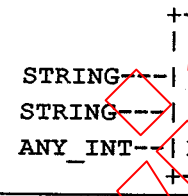
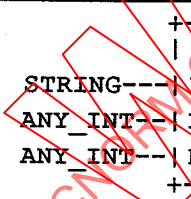
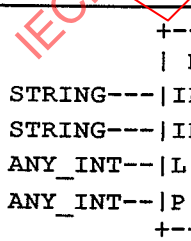
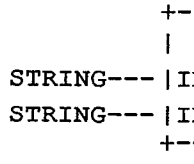
6 Usage examples and descriptions are given in the ST language defined in 3.3.

2.5.1.5.5 *Character string functions*

All the functions defined in 2.5.1.5.4 shall be applicable to character strings. For the purposes of comparison of two strings of unequal length, the shorter string shall be considered to be extended on the right to the length of the longer string by characters with the value zero. Comparison shall proceed from left to right, based on the numeric value of the character codes in the ISO/IEC 646 code table. For example, the character string 'Z' shall be greater than the character string 'AZ', and 'AZ' shall be greater than 'ABC'.

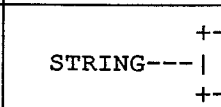
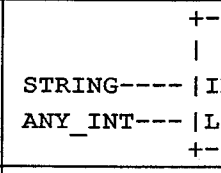
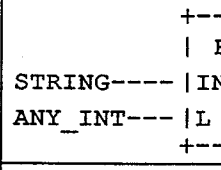
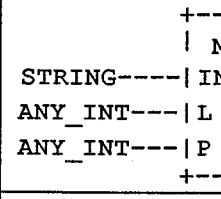
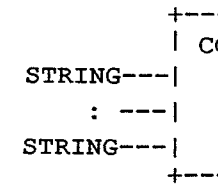
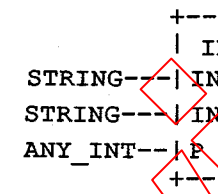
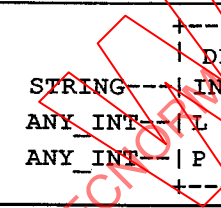
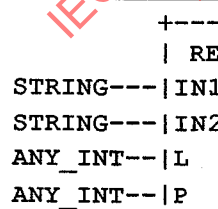
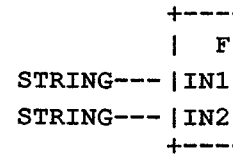
The standard graphical representations, function names and descriptions of additional functions of character strings shall be as shown in table 29. For the purpose of these operations, character positions within the string shall be considered to be numbered 1,2,...,L, beginning with the leftmost character position, where L is the length of the string.

Tableau 29 – Fonctions standards de cordons de caractères

N°	Forme graphique	Explication/exemple
1		Fonction longueur de cordon Exemple: A := LEN ('ASTRING') ; est équivalent à A := 7 ;
2		Caractères L les plus à gauche de IN Exemple: A := LEFT(IN := 'ASTR', L := 3) ; est équivalent à A := 'AST' ;
3		Caractères L les plus à droite de IN Exemple: A := RIGHT(IN := 'ASTR', L := 3) ; est équivalent à A := 'STR' ;
4		Les L caractères de IN, en commençant au P^{e} Exemple: A := MID(IN := 'ASTR', L := 2, P := 2) ; est équivalent à A := 'ST' ;
5		Enchaînement extensible Exemple: A := CONCAT ('AB', 'CD', 'E') ; est équivalent à A := 'ABCDE' ;
6		Insérer IN2 dans IN1 après la position du P^{e} caractère Exemple: A := INSERT(IN1 := 'ABC', IN2 := 'XY', P := 2) ; est équivalent à A := 'ABXYC' ;
7		Effacer L caractères de IN, en commençant à la position du P^{e} caractère Exemple: A := DELETE(IN := 'ABXYC', L := 2, P:= 3) ; est équivalent à A := 'ABC' ;
8		Remplacer L caractères de IN1 par IN2 en commençant à la position du P^{e} caractère Exemple: A := REPLACE(IN1 := 'ABCDE', IN2 := 'X', L := 2, P := 3) ; est équivalent à A := 'ABXE' ;
9		Trouver la position du caractère du début de la première apparition de IN2 dans IN1. Si aucune apparition de IN2 n'est trouvée, alors OUT := 0 Exemple: A := FIND(IN1 := 'ABCBC', IN2 := 'BC') ; est équivalent à A := 2 ;

NOTE - Les exemples fournis dans ce tableau sont exprimés dans le langage littéral structuré (ST) défini en 3.3.

Table 29 – Standard character string functions

No.	Graphical form	Explanation/example
1		String length function Example: A := LEN ('ASTRING') ; is equivalent to A := 7 ;
2		Leftmost L characters of IN Example: A := LEFT(IN := 'ASTR', L := 3) ; is equivalent to A := 'AST' ;
3		Rightmost L characters of IN Example: A := RIGHT(IN := 'ASTR', L := 3) ; is equivalent to A := 'STR' ;
4		L characters of IN, beginning at the P-th Example: A := MID(IN := 'ASTR', L := 2, P := 2) ; is equivalent to A := 'ST' ;
5		Extensible concatenation Example: A := CONCAT ('AB', 'CD', 'E') ; is equivalent to A := 'ABCDE' ;
6		Insert IN2 into IN1 after the P-th character position Example: A := INSERT(IN1 := 'ABC', IN2 := 'XY', P := 2) ; is equivalent to A := 'ABXYC' ;
7		Delete L characters of IN, beginning at the P-th character position Example: A := DELETE(IN := 'ABXYC', L := 2, P:=3) ; is equivalent to A := 'ABC' ;
8		Replace L characters of IN1 by IN2, starting at the P-th character position Example: A := REPLACE(IN1 := 'ABCDE', IN2 := 'X', L := 2, P := 3) ; is equivalent to A := 'ABXE' ;
9		Find the character position of the beginning of the first occurrence of IN2 in IN1. If no occurrence of IN2 is found, then OUT := 0 Example: A := FIND(IN1 := 'ABCBC', IN2 := 'BC') ; is equivalent to A := 2 ;
NOTE - The examples in this table are given in the Structured Text (ST) language defined in 3.3.		

2.5.1.5.6 *Fonctions de types de données relatifs au temps*

Outre les fonctions de comparaison et de sélection définies en 2.5.1.5.4, les combinaisons entre les types de données d'entrée et de sortie relatifs au temps, tels qu'indiqués au tableau 30, doivent être autorisées avec les fonctions associées.

2.5.1.5.7 *Fonctions des types de données énumérés*

Les fonctions de sélection et de comparaison indiquées au tableau 31 peuvent s'appliquer à des entrées appartenant à un type de donnée énuméré tel que défini en 2.3.3.1.

Tableau 30 – Fonctions des types de données relatifs au temps

Fonctions numériques et d'enchaînement					
N°	Nom	Symbole	IN1	IN2	OUT
1 2 3	ADD	+	TIME TIME_OF_DAY DATE_AND_TIME	TIME TIME TIME	TIME TIME_OF_DAY DATE_AND_TIME
4 5 6 7 8 9	SUB	–	TIME DATE TIME_OF_DAY TIME_OF_DAY DATE_AND_TIME DATE_AND_TIME	TIME DATE TIME TIME_OF_DAY TIME DATE_AND_TIME	TIME TIME TIME_OF_DAY TIME DATE_AND_TIME TIME
10 11	MUL DIV	* /	TIME TIME	ANY_NUM ANY_NUM	TIME TIME
12	CONCAT		DATE	TIME_OF_DAY	DATE_AND_TIME
	Fonctions de conversion type				
13 14	DATE_AND_TIME_TO_TIME_OF_DAY DATE_AND_TIME_TO_DATE				

NOTE – Les fonctions de conversion de types doivent avoir pour effet "l'extraction" des données appropriées, par exemple: les énoncés en langage ST

```
X := DT#1986-04-28-08:40:00 ;
Y := DATE_AND_TIME_TO_TIME_OF_DAY(X) ;
W := DATE_AND_TIME_TO_DATE(X)
```

doivent avoir les mêmes résultats que les énoncés

```
X := DT#1986-04-28-08:40:00 ;
W := DATE#1986-04-28 ;
Y := TIME_OF_DAY#08:40:00
```

Tableau 31 – Fonctions de types de données énumérés

N°	Nom	Symbole	Numéro de caractéristique en 2.5.1.5.4
1	SEL		1
2	MUX		4
3	EQ	=	7
4	NE	<>	10

2.5.1.5.6 *Functions of time data types*

In addition to the comparison and selection functions defined in 2.5.1.5.4, the combinations of input and output time data types shown in table 30 shall be allowed with the associated functions.

2.5.1.5.7 *Functions of enumerated data types*

The selection and comparison functions listed in table 31 can be applied to inputs which are of an enumerated data type as defined in 2.3.3.1.

Table 30 – Functions of time data types

Numeric and concatenation functions					
No.	Name	Symbol	IN1	IN2	OUT
1 2 3	ADD	+	TIME TIME_OF_DAY DATE_AND_TIME	TIME TIME TIME	TIME TIME_OF_DAY DATE_AND_TIME
4 5 6 7 8 9	SUB	–	TIME DATE TIME_OF_DAY TIME_OF_DAY DATE_AND_TIME DATE_AND_TIME	TIME DATE TIME TIME_OF_DAY TIME DATE_AND_TIME	TIME TIME TIME_OF_DAY TIME DATE_AND_TIME TIME
10 11	MUL DIV	* /	TIME TIME	ANY_NUM ANY_NUM	TIME TIME
12	CONCAT		DATE	TIME_OF_DAY	DATE_AND_TIME
	Type conversion functions				
13 14	DATE_AND_TIME_TO_TIME_OF_DAY DATE_AND_TIME_TO_DATE				

NOTE - The type conversion functions shall have the effect of "extracting" the appropriate data, e.g., the ST language statements

```
X := DT#1986-04-28-08:40:00 ;
Y := DATE_AND_TIME_TO_TIME_OF_DAY(X) ;
W := DATE_AND_TIME_TO_DATE(X)
```

shall have the same result as the statements

```
X := DT#1986-04-28-08:40:00 ;
W := DATE#1986-04-28 ;
Y := TIME_OF_DAY#08:40:00
```

Table 31 – Functions of enumerated data types

No.	Name	Symbol	Feature number in 2.5.1.5.4
1	SEL		1
2	MUX		4
3	EQ	=	7
4	NE	<>	10

2.5.2 *Blocs fonctionnels*

Pour les besoins des langages de programmation d'automates programmables, un *bloc fonctionnel* est une unité d'organisation de programme qui, lorsqu'elle est exécutée, donne une ou plusieurs valeurs. Il est possible de créer plusieurs *instances* (copies) nommées d'un bloc fonctionnel. A chaque instance doit être associé un identificateur (*le nom de l'instance),* une structure de données contenant ses variables internes et ses variables de sortie, et, en fonction de l'application concernée, des valeurs ou des références relatives à ses paramètres d'entrée. Toutes les valeurs des variables de sortie et des variables internes nécessaires de cette structure de données doivent persister, d'un bloc fonctionnel au suivant; par conséquent, il n'est pas toujours nécessaire que le lancement du même bloc fonctionnel ayant les mêmes arguments (paramètres d'entrée) aboutisse aux mêmes valeurs de sortie.

Seuls les paramètres d'entrée et de sortie doivent être accessibles à l'extérieur de l'instance d'un bloc fonctionnel, c'est-à-dire que les variables internes du bloc fonctionnel doivent être cachées à l'utilisateur du bloc fonctionnel.

L'exécution des opérations d'un bloc fonctionnel doit être lancée conformément aux prescriptions de l'article 3 relatif au langages littéraux; cette exécution doit être lancée conformément aux règles d'évaluation de réseau, données à l'article 4 pour les langages graphiques, ou sous le contrôle d'éléments de commande d'exécution tels que définis en 2.6.

Tout bloc fonctionnel qui a déjà été déclaré, peut être utilisé dans la déclaration d'un autre bloc fonctionnel ou d'un autre programme, comme illustré à la figure 3.

Le champ d'application d'une instance de bloc fonctionnel doit être local à l'unité d'organisation de programme dans laquelle il est instancié, sauf si cette instance est déclarée comme globale dans un bloc VAR_GLOBAL tel que défini en 2.7.1.

Comme illustré en 2.5.2.2, le nom d'instance d'une instance de bloc fonctionnel peut être utilisé comme l'entrée d'une fonction ou d'un bloc fonctionnel, si cette instance est déclarée comme une variable d'entrée dans une déclaration VAR_INPUT, ou comme une variable d'entrée/sortie de bloc fonctionnel dans une déclaration VAR_IN_OUT, comme cela est défini en 2.4.3.

2.5.2.1 *Représentation*

Comme l'illustre la figure 9, une instance de bloc fonctionnel peut être créée *littéralement,* en déclarant un élément de donnée en utilisant le type de bloc de fonction déclaré dans une construction VAR...END_VAR, de façon identique à l'utilisation d'un type de donnée structurée, tel que défini en 3.

Comme l'illustre la figure 9 ci-dessous, l'instance d'un bloc fonctionnel peut être créée *graphiquement,* en utilisant une représentation graphique du bloc fonctionnel, avec le nom de type de bloc fonctionnel à l'intérieur du bloc, et le nom de l'instance au-dessus du bloc, conformément aux règles relatives à la représentation de fonctions données en 2.5.1.1, avec les conditions supplémentaires suivantes:

1) La taille et l'orientation du bloc peuvent varier selon le nombre d'entrées, de sorties et suivant le nombre d'informations qui doivent être affichées.

2) Les noms des paramètres formels d'entrée et de sortie doivent respectivement apparaître à l'intérieur des côtés gauche et droit d'un bloc.

2.5.2 *Function blocks*

For the purposes of programmable controller programming languages, a *function block* is a program organization unit which, when executed, yields one or more values. Multiple, named *instances* (copies) of a function block can be created. Each instance shall have an associated identifier (the *instance name*), and a data structure containing its output and internal variables, and, depending on the implementation, values of or references to its input parameters. All the values of the output variables and the necessary internal variables of this data structure shall persist from one execution of the function block to the next; therefore, invocation of the same function block with the same arguments (input parameters) need not always yield the same output values.

Only the input and output parameters shall be accessible outside of an instance of a function block, i.e., the function block's internal variables shall be hidden from the user of the function block.

Execution of the operations of a function block shall be invoked as defined in clause 3 for textual languages, according to the rules of network evaluation given in clause 4 for graphic languages, or under the control of sequential function chart (SFC) elements as defined in 2.6.

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.

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.

As illustrated in 2.5.2.2, the instance name of a function block instance can be used as the input to a function or function block if declared as an input variable in a VAR_INPUT declaration, or as an input/output variable of a function block in a VAR_IN_OUT declaration, as defined in 2.4.3.

2.5.2.1 *Representation*

As illustrated in figure 9, an instance of a function block can be created *textually*, by declaring a data element using the declared function block type in a VAR...END_VAR construct, identically to the use of a structured data type, as defined in 2.4.3.

As further illustrated in figure 9, an instance of a function block can be created *graphically*, by using a graphic representation of the function block, with the function block type name inside the block, and the instance name above the block, following the rules for representation of functions given in 2.5.1.1 with the following additional conditions:

1) The size and orientation of the block may vary depending on the number of inputs, outputs, and other information to be displayed.

2) Formal input and output parameter names shall be shown at the inside left and right sides of the block, respectively.

Comme l'illustre la figure 9, les variables d'entrée et de sortie d'une instance de bloc fonctionnel peuvent être représentées comme des éléments de types de données structurés, tels que définis en 2.3.6.1.

Si l'une des deux caractristiques d'inversion graphiques définies au tableau 19 est acceptée pour des blocs fonctionnels, elle doit être également acceptée pour les fonctions définies en 2.5.1 et réciproquement.

Les instances de blocs fonctionnels peuvent être déclarées comme non volatiles; se reporter à la caractéristiques n° 3 du tableau 33.

Graphique (langage FBD)	Littéral (langage ST)

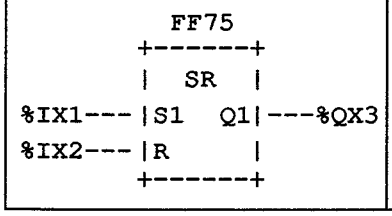
	`VAR FF75 : SR ; END_VAR (* Déclaration *)`
	`FF75 (S1 := %IX1, R := %IX2); (* Lancement *)`
	`%QX3 := FF75.Q1 ; (* Sortie d'affectation *)`

Figure 9 – Exemple d'instanciation de bloc fonctionnel

L'affectation d'une valeur à une variable de sortie de bloc foctionnel ne doit pas être autorisée, sauf à l'intérieur d'un bloc fonctionnel. L'affectation d'une valeur à une entrée de bloc fonctionnel n'est admise que si elle fait partie du lancement du bloc fonctionnel. Les utilisations autorisées des entrées et des sorties de blocs fonctionnels sont résumées au tableau 32, en utilisant le bloc fonctionnel FF75 de type SR illustré dans la figure 9. Les exemples illustrés sont exprimés en langage ST.

Tableau 32 – Exemples d'utilisation d'un paramètre d'E/S de bloc fonctionnel

Utilisation	A l'intérieur du bloc fonctionnel	A l'extérieur du bloc fonctionnel
Input Read	IF S1 THEN	Non autorisé (notes 1 et 2)
Input Write	Non autorisé (notes 1 et 3)	FF75(S1 := %IX1, R := %IX2) ;
Output Read	Q1 := Q1 AND NOT R ;	%QX3 := FF75.Q1 ;
Output Write	Q1 := 1 ;	Non autorisé (note 1)

NOTES

1 Les applications qualifiées de "non autorisées" dans le présent tableau, peuvent avoir des conséquences imprévisibles propres à l'application concernée.

2 La lecture d'une entrée de bloc fonctionnel peut être effectuée par la "fonction de communication", la fonction d'"interface opérateur", ou par les "fonctions de programmation, de test et de surveillance" définies dans la première partie de la présente norme.

3 Comme l'illustre la figure 2.5.2.2, une modification dans le bloc fonctionnel d'une variable déclarée dans un bloc VAR_IN_OUT est autorisée.

As shown in figure 9, input and output variables of an instance of a function block can be represented as elements of structured data types as defined in 2.3.6.1.

If either of the two graphical negation features defined in table 19 is supported for function blocks, it shall also be supported for functions as defined in 2.5.1, and vice versa.

Function block instances can be declared to be retentive, as shown in feature 3 of table 33.

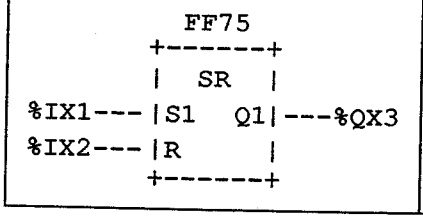
Graphical (FBD language)	Textual (ST language)
	VAR FF75 : SR ; END_VAR (* Declaration *) FF75(S1 := %IX1, R := %IX2) ; (* Invocation *) %QX3 := FF75.Q1 ; (* Assign Output *)

Figure 9 – Function block instantiation example

Assignment of a value to an output variable of a function block is not allowed except from within the function block. The assignment of a value to the input of a function block is permitted only as part of the invocation of the function block. Allowable usages of function block inputs and outputs are summarized in table 32, using the function block FF75 of type SR shown in figure 9. The examples are shown in the ST language.

Table 32 – Examples of function block I/O parameter usage

Usage	Inside function block	Outside function block
Input Read	IF S1 THEN	Not allowed (note 1 and 2)
Input Write	Not allowed (notes 1 and 3)	FF75(S1 := %IX1, R := %IX2) ;
Output Read	Q1 := Q1 AND NOT R ;	%QX3 := FF75.Q1 ;
Output Write	Q1 := 1 ;	Not allowed (note 1)

NOTES

1 Those usages listed as "not allowed" in this table could lead to implementation-dependent, unpredictable side effects.

2 Reading of an input of a function block may be performed by the "communication function", "operator interface function", or the "programming, testing, and monitoring functions" defined in part 1 of this standard.

3 As illustrated in 2.5.2.2, modification within the function block of a variable declared in a VAR_IN_OUT block is permitted.

2.5.2.2 *Déclaration*

Comme l'illustre la figure 10, un bloc fonctionnel doit être déclaré littéralement ou graphiquement, de la même manière que celle définie pour les fonctions décrites en 2.5.1.3, avec les différences décrites ci-après et résumées au tableau 33.

1) Les mots clés de séparation, relatifs à la déclaration de blocs fonctionnels, doivent être comme suit: FUNCTION_BLOCK...END_FUNCTION_BLOCK.

2) Un bloc fonctionnel peut avoir plus d'un paramètre de sortie, déclaré littéralement avec la construction VAR_OUTPUT...END_VAR définie en 2.4.3, ou graphiquement comme l'illustre la figure 10.

3) Le qualificatif RETAIN, défini en 2.4.3, peut être utilisé pour des variables internes et de sortie de bloc fonctionnel, comme l'indiquent les caractéristiques n° 1, n° 2, et n° 3 du tableau 33.

4) Les valeurs de variables, qui sont transmises au bloc fonctionnel par l'intermédiaire d'une construction VAR_IN_OUT ou VAR_EXTERNAL, peuvent être modifiées de l'intérieur du bloc fonctionnel, comme l'illustre la caractéristique n° 4 du tableau 33.

5) Comme l'illustrent les caractéristiques n° 5, n° 6, et n° 7 du tableau 33, il est possible, d'accéder à, mais non de modifier, les valeurs de sortie d'une instance de bloc fonctionnel dont le nom est transféré dans le bloc fonctionnel par l'intermédiaire d'une construction VAR_INPUT, VAR_IN_OUT ou VAR_EXTERNAL, à partir du bloc fonctionnel.

6) Un bloc fonctionnel, dont le nom d'instance est transféré dans le bloc fonctionnel par l'intermédiaire d'une construction VAR_IN_OUT ou VAR_EXTERNAL, peut être lancé de l'intérieur du bloc fonctionnel, comme l'illustrent les caractéristiques n° 6 et n° 7 du tableau 33.

7) Dans des déclarations littérales, les qualificatifs R_EDGE et F_EDGE peuvent être utilisés pour indiquer une fonction de détection de fronts sur les entrées booléennes. Ils doivent entraîner la déclaration implicite d'un bloc fonctionnel de type R_TRIG ou F_TRIG, respectivement, conformément en 2.5.2.3.2, pour effectuer la détection requise de fronts. En ce qui concerne un exemple de cette construction, se reporter aux caractéristiques n^{os} 8a et 8b du tableau 33 et à la note d'accompagnement.

8) La construction illustrée au tableau 33, ainsi que les caractéristiques n^{os} 9a et 9b doivent être utilisées dans des déclarations graphiques pour la détection de fronts montants et de fronts descendants. Lorsque le jeu de caractères ISO/IEC 646 est utilisé, le caractère "supérieur à" (>) ou le caractère "inférieur à" (<) doit être en ligne avec le bord du bloc fonctionnel. Lorsque des représentations graphiques ou semi-graphiques sont employées, il est nécessaire d'utiliser la notation de la CEI 617-12, relative au entrées dynamiques.

9) Les constructions d'initialisation de variables, définies en 2.4.3.2, peuvent être utilisées pour la déclaration des valeurs par défaut des entrées de blocs fonctionnels, ainsi que pour celle des valeurs initiales de leurs variables internes et de leurs variables de sortie.

Comme l'illustre la figure 12, seuls les noms de variables ou d'instances de blocs fonctionnels peuvent être transférés dans un bloc fonctionnel par l'intermédiaire de la construction VAR_IN_OUT, c'est-à-dire que les sorties de fonctions ou de blocs fonctionnels ne peuvent être transférées par l'intermédiaire de cette construction. Ceci a pour but d'éviter des modifications involontaires de telles sorties. Cependant, l'arrangement "en cascade" des constructions VAR_IN_OUT est autorisé, comme l'illustre la figure 12c.

2.5.2.2 *Declaration*

As illustrated in figure 10, a function block shall be declared textually or graphically in the same manner as defined for functions in 2.5.1.3, with the differences described below and summarized in table 33.

1) The delimiting keywords for declaration of function blocks shall be FUNCTION_BLOCK...END_FUNCTION_BLOCK.

2) A function block can have more than one output parameter, declared textually with the VAR_OUTPUT...END_VAR construct defined in 2.4.3, or graphically as illustrated in figure 10.

3) The RETAIN qualifier defined in 2.4.3 can be used for internal and output variables of a function block, as shown in features 1, 2, and 3 in table 33.

4) The values of variables which are passed to the function block via a VAR_IN_OUT or VAR_EXTERNAL construct can be modified from within the function block, as shown in feature 4 of table 33.

5) The output values of a function block instance whose name is passed into the function block via a VAR_INPUT, VAR_IN_OUT, or VAR_EXTERNAL construct can be accessed, but not modified, from within the function block, as shown in features 5, 6, and 7 of table 33.

6) A function block whose instance name is passed into the function block via a VAR_IN_OUT or VAR_EXTERNAL construction can be invoked from inside the function block, as shown in features 6 and 7 of table 33.

7) In textual declarations, the R_EDGE and F_EDGE qualifiers can be used to indicate an edge-detection function on Boolean inputs. This shall cause the implicit declaration of a function block of type R_TRIG or F_TRIG, respectively, as defined in 2.5.2.3.2, to perform the required edge detection. For an example of this construction, see features 8a and 8b of table 33 and the accompanying NOTE.

8) The construction illustrated in table 33, features 9a and 9b shall be used in graphical declarations for rising and falling edge detection. When the ISO/IEC 646 character set is used, the "greater than" (>) or "less than" (<) character shall be in line with the edge of the function block. When graphic or semigraphic representations are employed, the notation of IEC 617, part 12 for dynamic inputs shall be used.

9) The variable initialization constructs defined in 2.4.3.2 can be used for the declaration of default values of function block inputs and initial values of their internal and output variables.

As illustrated in figure 12, only variables or function block instance names can be passed into a function block via the VAR_IN_OUT construct, i.e., function or function block outputs cannot be passed via this construction. This is to prevent the inadvertent modifications of such outputs. However, "cascading" of VAR_IN_OUT constructions is permitted, as illustrated in figure 12c.

a)

```
FUNCTION_BLOCK DEBOUNCE
(*** Interface externe ***)

  VAR_INPUT
    IN : BOOL ;                      (* Par défaut = 0 *)
    DB_TIME : TIME := t#10ms ;       (* Par défaut = t#10ms *)
  END_VAR
  VAR_OUTPUT OUT : BOOL ;            (* Par défaut = 0 *)
      ET_OFF : TIME ;                (* Par défaut = t#0s *)
  END_VAR
  VAR DB_ON : TON ;                  (** Variables internes **)
      DB_OFF : TON ;                 (** et instances FB **)
      DB_FF : SR ;
  END_VAR

(** Corps de bloc fonctionnel **)

DB_ON(IN := IN, PT := DB_TIME) ;
DB_OFF(IN := NOT IN, PT := DB_TIME) ;
DB_FF(S1 := DB_ON.Q, R := DB_OFF.Q) ;
OUT := DB_FF.Q ;
ET_OFF := DB_OFF.ET ;
END_FUNCTION_BLOCK
```

b)

```
FUNCTION_BLOCK
(** Interface externe **)

                        +---------------+
                        |   DEBOUNCE    |
                  BOOL---|IN         OUT|---BOOL
                  TIME---|DB_TIME ET_OFF|---TIME
                        +---------------+

(** Corps de bloc fonctionnel **)

                          DB_ON        DB_FF
                         +-----+      +-----+
                         | TON |      | SR  |
                IN----+--|IN  Q|------|S1  Q|---OUT
                      |  |PT ET|   +--|R    |
                      |  +-----+   |  +-----+
                      |            |
                      |            |
                      |   DB_OFF   |
                      |  +-----+   |
                      |  | TON |   |
                      +--|--O|IN  Q|--+
                         |PT ET|
                DB_TIME--+-----+--------------ET_OFF

END_FUNCTION_BLOCK
```

Figure 10 – Exemples de déclarations de blocs fonctionnels
a) Déclaration littérale en langage ST (voir 3.3)
b) Déclaration graphique en langage FBD (voir 4.3)

a)

```
FUNCTION_BLOCK DEBOUNCE
(*** External Interface ***)
  VAR_INPUT
    IN : BOOL ;                    (* Default = 0 *)
    DB_TIME : TIME := t#10ms ;     (* Default = t#10ms *)
  END_VAR
  VAR_OUTPUT OUT : BOOL ;          (* Default = 0 *)
      ET_OFF : TIME ;              (* Default = t#0s *)
  END_VAR
  VAR DB_ON : TON ;                (** Internal Variables **)
      DB_OFF : TON ;               (** and FB Instances **)
      DB_FF : SR ;
  END_VAR
(** Function Block Body**)
DB_ON(IN := IN, PT := DB_TIME) ;
DB_OFF(IN := NOT IN, PT := DB_TIME) ;
DB_FF(S1 := DB_ON.Q, R := DB_OFF.Q) ;
OUT := DB_FF.Q ;
ET_OFF := DB_OFF.ET ;
END_FUNCTION_BLOCK
```

b)

```
FUNCTION_BLOCK
(** External Interface **)
                          +----------------+
                          |    DEBOUNCE    |
                   BOOL---|IN          OUT |---BOOL
                   TIME---|DB_TIME  ET_OFF |---TIME
                          +----------------+
(** Function Block Body **)
                            DB_ON        DB_FF
                           +-----+      +-----+
                           | TON |      | SR  |
                     IN----+-----|IN  Q|-----|S1  Q|---OUT
                           |  +--|PT ET|  +--|R    |
                           |  |  +-----+  |  +-----+
                           |  |           |
                           |  |  DB_OFF   |
                           |  |  +-----+  |
                           |  |  | TON |  |
                           +--|--O|IN  Q|--+
                      DB_TIME--+---|PT ET|--------------ET_OFF
                               +-----+
END_FUNCTION_BLOCK
```

Figure 10 – Examples of function block declarations
a) Textual declaration in ST language (see 3.3)
b) Graphical declaration in FBD language (see 4.3)

Tableau 33 – Caractéristiques des déclarations de blocs fonctionnels

N°	Description	Exemple
1	Qualificatif RETAIN sur des variables internes	VAR RETAIN X : REAL ; END_VAR
2	Qualificatif RETAIN sur des variables de sortie	VAR_OUTPUT RETAIN X : REAL ; END_VAR
3	Qualificatif RETAIN sur des blocs fonctionnels internes	VAR RETAIN TMR1 : TON ; END_VAR
4a	Déclaration d'entrée/sortie (littérale)	VAR_INPUT X : INT ; END_VAR VAR_IN_OUT A : INT ; END_VAR A := A+X ;
4b	Déclaration d'entrée/sortie (graphique)	Voir figure 12
5a	Nom d'instance de bloc fonctionnel comme entrée (littéral)	VAR_INPUT I_TMR : TON ; END_VAR EXPIRED := I_TMR.Q ; (* Note 1 *)
5b	Nom d'instance de bloc fonctionnel comme entrée (graphique)	Voir figure 11a
6a	Nom d'instance de bloc fonctionnel comme entrée/sortie (littéral)	VAR_IN_OUT IO_TMR : TOF ; END_VAR IO_TMR(IN := A_VAR, PT := T#10S) ; EXPIRED := I_TMR.Q ; (* Note 1 *)
6b	Nom d'instance de bloc fonctionnel comme entrée/sortie (graphique)	Voir figure 11b
7a	Nom d'instance de bloc fonctionnel comme variable externe (littéral)	VAR_EXTERNAL EX_TMR : TOF ; END_VAR EX_TMR(IN := A_VAR, PT := T#10S) ; EXPIRED := EX_TMR.Q ; (* Note 1 *)
7b	Nom d'instance de bloc fonctionnel comme variable externe (graphique)	Voir figure 11c
8a 8b	Déclaration littérale: entrée de front montant entrée de front descendant	FUNCTION_BLOCK AND_EDGE (* Note 2 *) VAR_INPUT X : BOOL R_EDGE ; Y : BOOL F_EDGE ; END_VAR VAR_OUTPUT Z : BOOL ; END_VAR Z := X AND Y ; (* Exemple en langage ST *) END_FUNCTION_BLOCK (* - voir 3.3 *)
9a 9b	Déclaration graphique: entrée de front montant entrée de front descendant	FUNCTION_BLOCK (* Note 2 *) +-----------+ (* Interface externe *) | AND_EDGE | BOOL---->X Z|---BOOL | | BOOL---<Y | | | +-----------+ +---+ (* Corps de bloc fonctionnel *) X---| & |---Z (* Exemple de langage FBD *) Y---| | (* - Voir 4.3 *) +---+ END_FUNCTION_BLOCK

NOTES

1 On suppose dans ces exemples que les variables EXPIRED et A_VAR ont été déclarées comme du type BOOL.

2 La déclaration du bloc fonctionnel AND_EDGE, dans les exemples ci-dessus, est équivalent à:

```
FUNCTION_BLOCK AND_EDGE
  VAR_INPUT XCLK : BOOL ; YCLK : BOOL ; END_VAR
  VAR X_TRIG : R_TRIG ; Y_TRIG : F_TRIG ; END_VAR
  X_TRIG(CLK := XCLK) ; X := X_TRIG.Q ;
  Y_TRIG(CLK := YCLK) ; Y := Y_TRIG.Q ;
  Z := X AND Y ;
END_FUNCTION_BLOCK
```

Voir 2.5.2.3.2, pour la définition des blocs fonctionnels de détection de fronts R_TRIG et F_TRIG.

Table 33 – Function block declaration features

No.	Description	Example
1	RETAIN qualifier on internal variables	VAR RETAIN X : REAL ; END_VAR
2	RETAIN qualifier on output variables	VAR_OUTPUT RETAIN X : REAL ; END_VAR
3	RETAIN qualifier on internal function blocks	VAR RETAIN TMR1 : TON ; END_VAR
4a	Input/output declaration (textual)	VAR_INPUT X : INT ; END_VAR VAR_IN_OUT A : INT ; END_VAR A := A+X ;
4b	Input/output declaration (graphical)	See figure 12
5a	Function block instance name as input (textual)	VAR_INPUT I_TMR : TON ; END_VAR EXPIRED := I_TMR.Q ; (* Note 1 *)
5b	Function block instance name as input (graphical)	See figure 11a
6a	Function block instance name as input/output (textual)	VAR_IN_OUT IO_TMR : TOF ; END_VAR IO_TMR(IN := A_VAR, PT := T#10S) ; EXPIRED := I_TMR.Q ; (* Note 1 *)
6b	Function block instance name as input/output (graphical)	See figure 11b
7a	Function block instance name as external variable (textual)	VAR_EXTERNAL EX_TMR : TOF ; END_VAR EX_TMR(IN := A_VAR, PT := T#10S) ; EXPIRED := EX_TMR.Q ; (* Note 1 *)
7b	Function block instance name as external variable (graphical)	See figure 11c
8a 8b	Textual declaration of: rising edge inputs falling edge inputs	FUNCTION_BLOCK AND_EDGE (* Note 2 *) VAR_INPUT X : BOOL R_EDGE ; Y : BOOL F_EDGE ; END_VAR VAR_OUTPUT Z : BOOL ; END_VAR Z := X AND Y ; (* ST language example *) END_FUNCTION_BLOCK (* - see 3.3 *)
9a 9b	Graphical declaration of: rising edge inputs falling edge inputs	FUNCTION_BLOCK (* Note 2 *) +----------+ (* External interface *) | AND_EDGE | BOOL---->X Z|---BOOL | | BOOL---<Y | | | +----------+ +---+ (* Function block body *) X---| & |---Z (* FBD language example *) Y---| | (* - see 4.3 *) +---+ END_FUNCTION_BLOCK

NOTES

1 It is assumed in these examples that the variables EXPIRED and A_VAR have been declared of type BOOL.

2 The declaration of function block AND_EDGE in the above examples is equivalent to:

```
FUNCTION_BLOCK AND_EDGE
  VAR_INPUT XCLK : BOOL ; YCLK : BOOL ; END_VAR
  VAR X_TRIG : R_TRIG ; Y_TRIG : F_TRIG ; END_VAR
  X_TRIG(CLK := XCLK) ; X := X_TRIG.Q ;
  Y_TRIG(CLK := YCLK) ; Y := Y_TRIG.Q ;
  Z := X AND Y ;
END_FUNCTION_BLOCK
```

See 2.5.2.3.2 for the definition of the edge detection function blocks R_TRIG and F_TRIG.

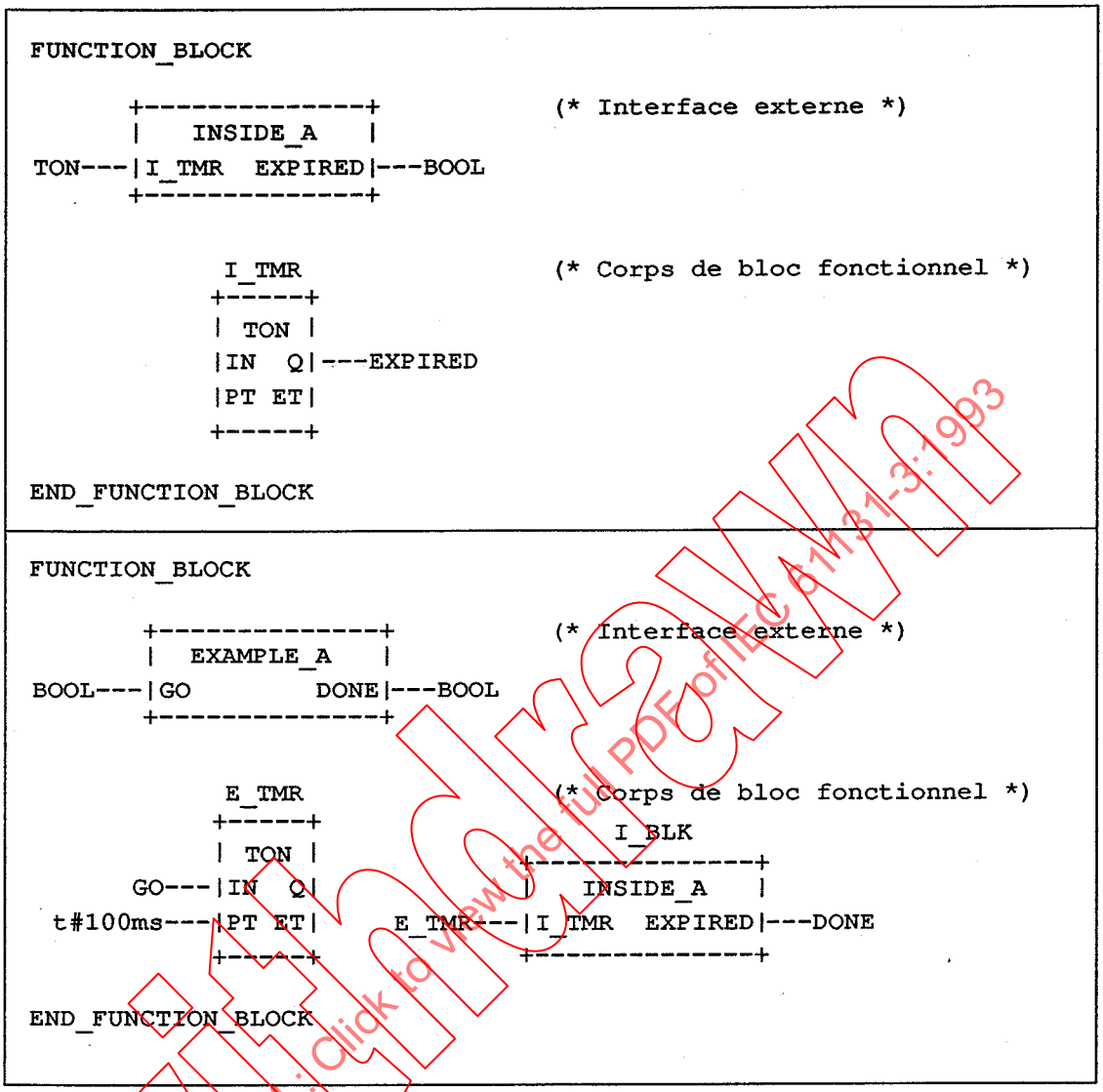


Figure 11a – Utilisation graphique d'un nom de bloc fonctionnel, en tant que variable d'entrée (tableau 33, caractéristique N° 5b)

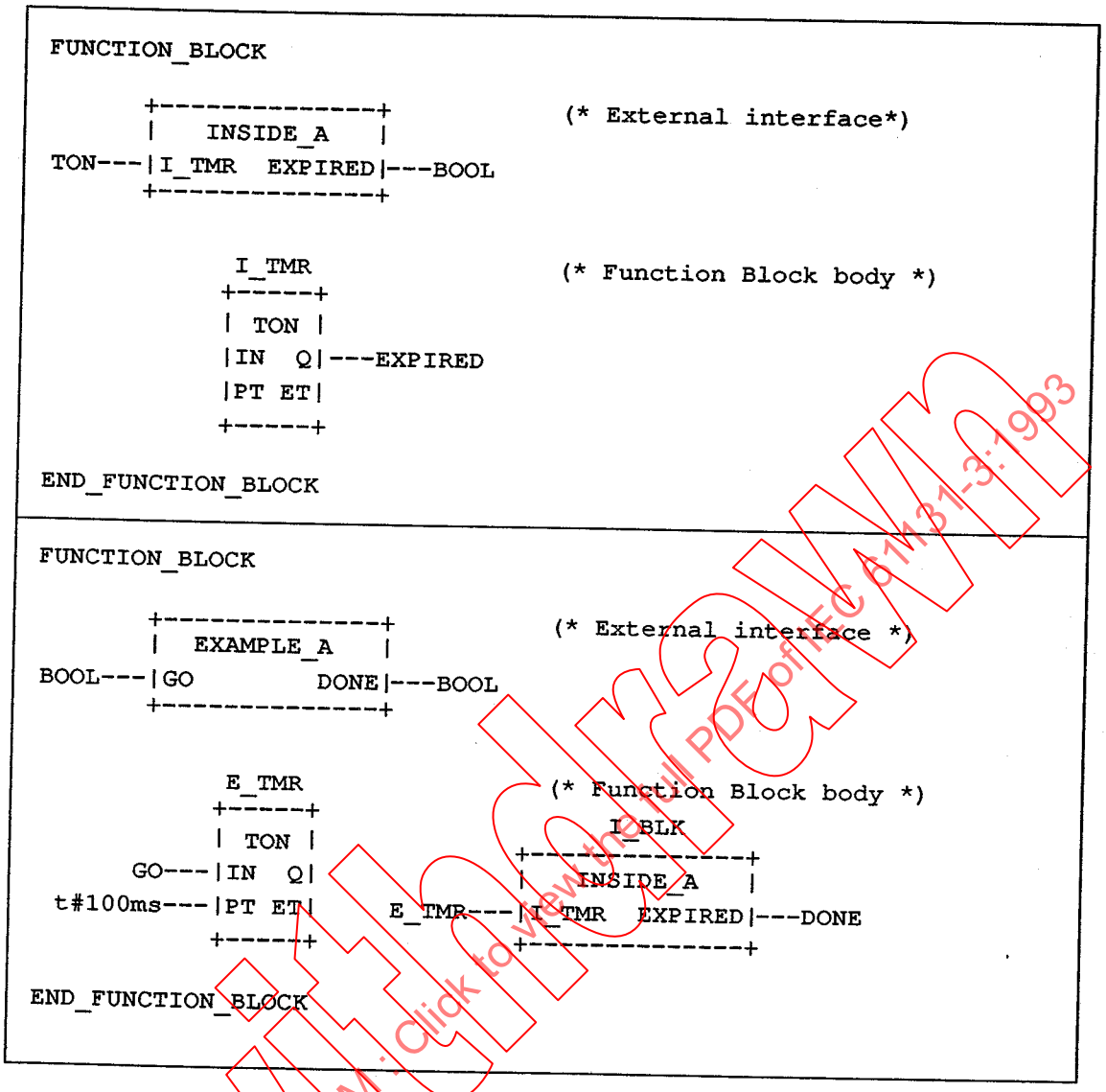


Figure 11a – Graphical use of a function block name as an input variable (table 33, feature 5b)

```
FUNCTION_BLOCK

      +---------------+              (* Interface externe *)
      |    INSIDE_B   |
TON---|I_TMR     I_TMR|---TON
BOOL--|TMR_GO EXPIRED|---BOOL
      +---------------+


            I_TMR                    (* Corps de bloc fonctionnel *)
           +-----+
           | TON |
 TMR_GO---|IN  Q|---EXPIRED
           |PT ET|
           +-----+

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK

       +---------------+             (* Interface externe *)
       |  EXAMPLE_B    |
BOOL---|GO          DONE|---BOOL
       +---------------+

                                     (* Corps de bloc fonctionnel *)
             E_TMR                        I_BLK
            +-----+                +---------------+
            | TON |                |   INSIDE_B    |
            |IN  Q|          E_TMR---|I_TMR-----I_TMR|
 t#100ms----|PT ET|             GO------|TMR_GO  EXPIRED|---DONE
            +-----+                +---------------+

END_FUNCTION_BLOCK
```


Figure 11b – Utilisation graphique d'un nom de bloc fonctionnel, en tant que variable d'entrée/sortie (tableau 33, caractéristique n° 6b)

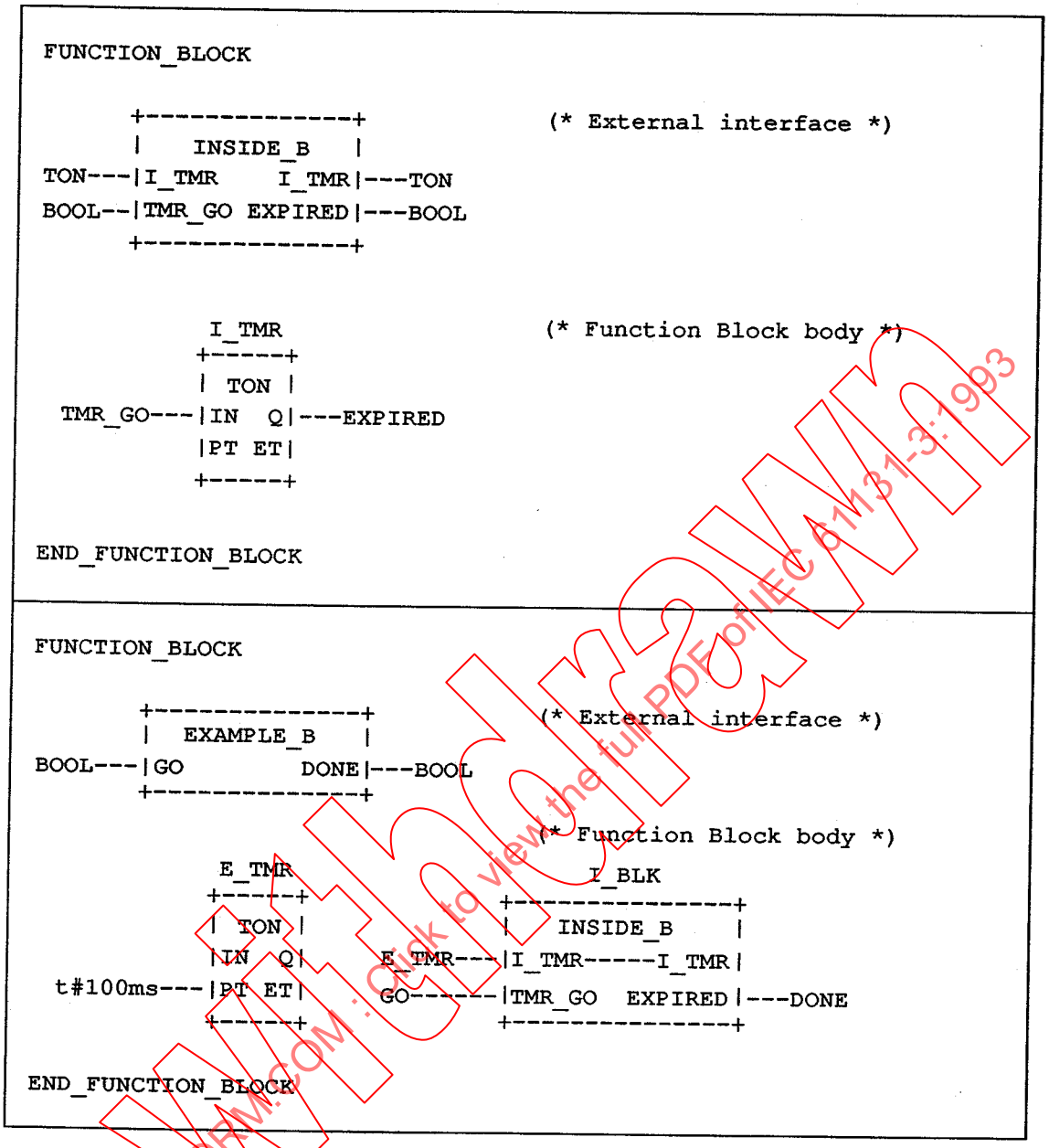
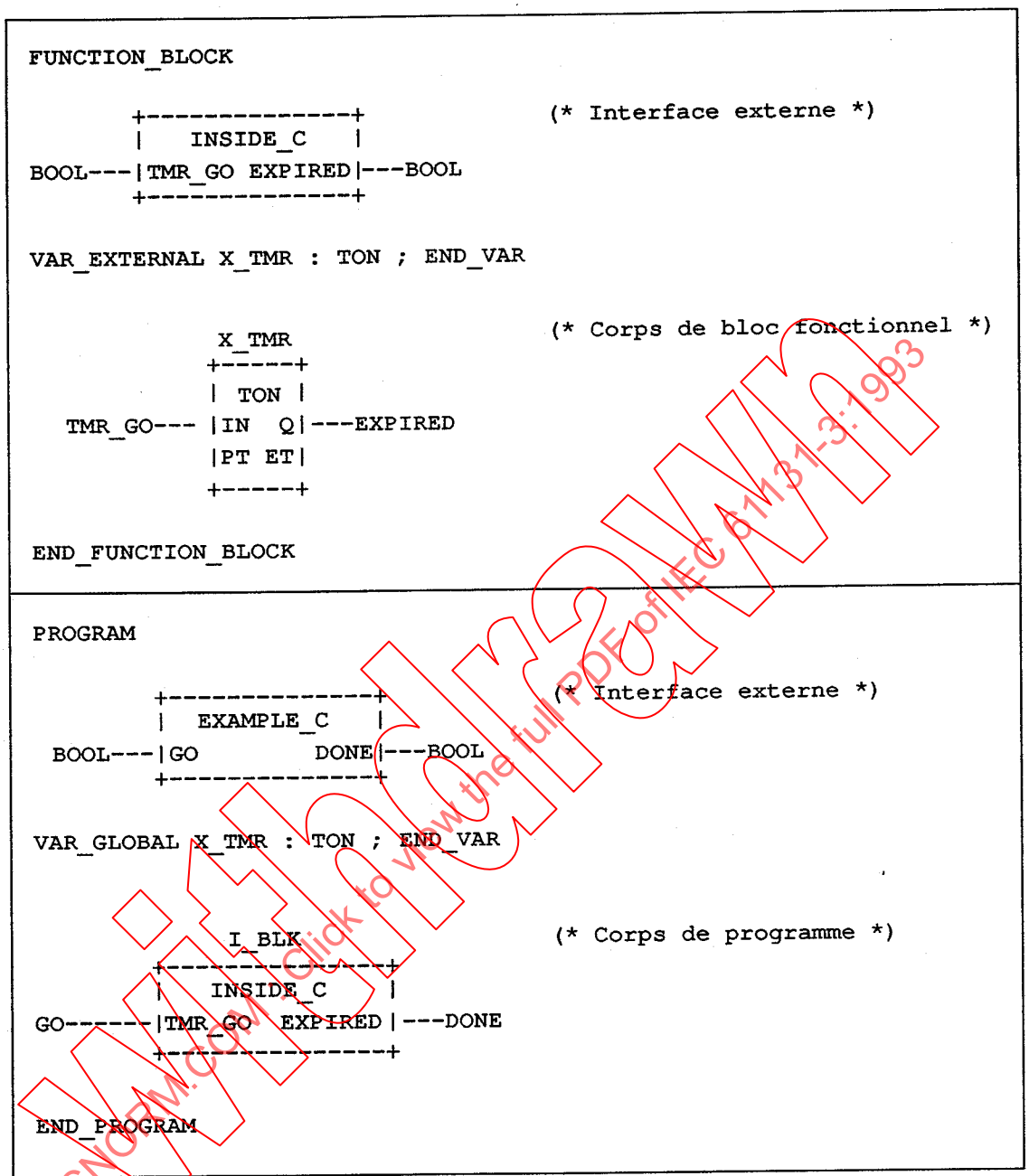
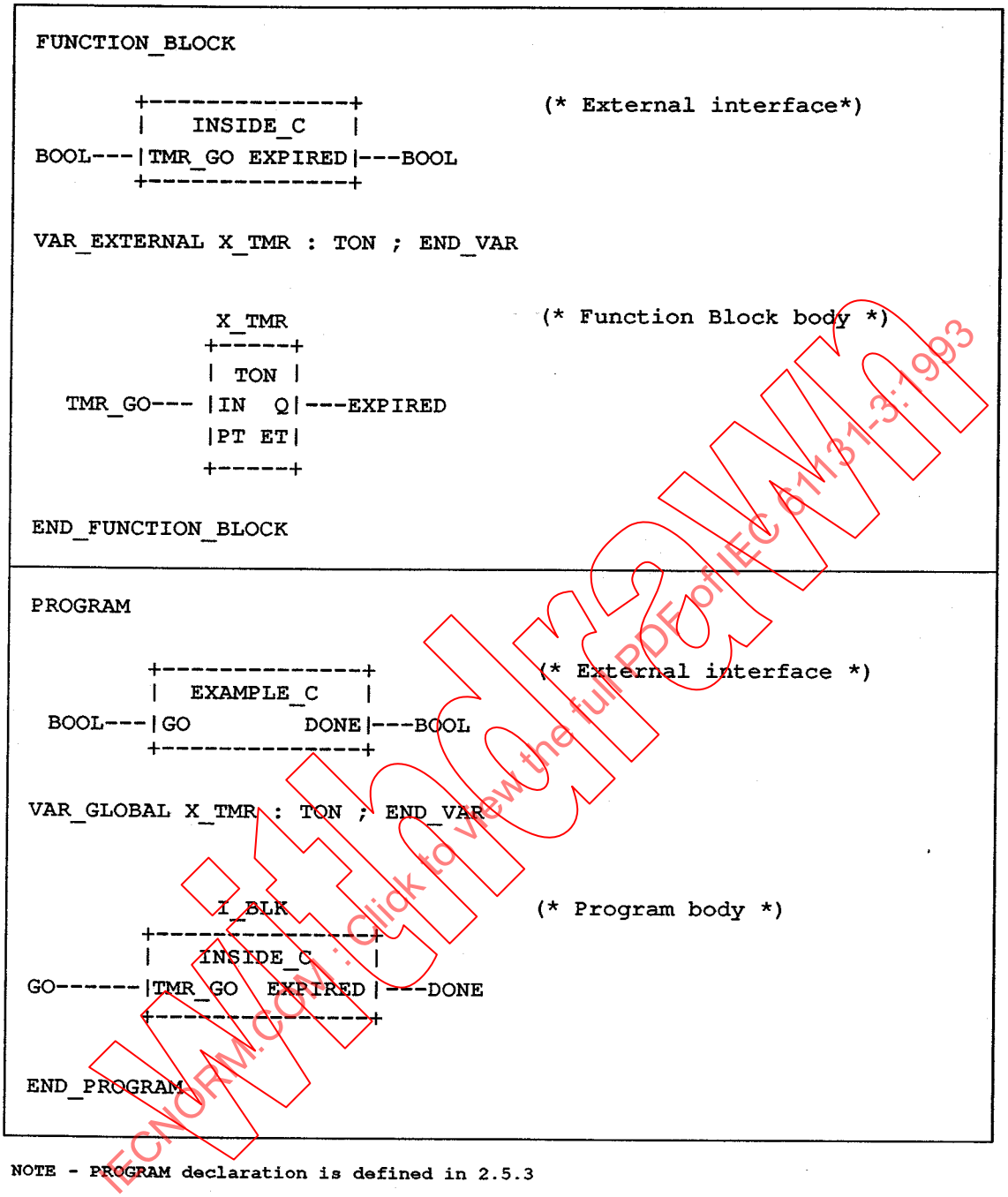


Figure 11b – Graphical use of a function block name as an input/output variable (table 33, feature 6b)



NOTE – La déclaration PROGRAM est définie en 2.5.3

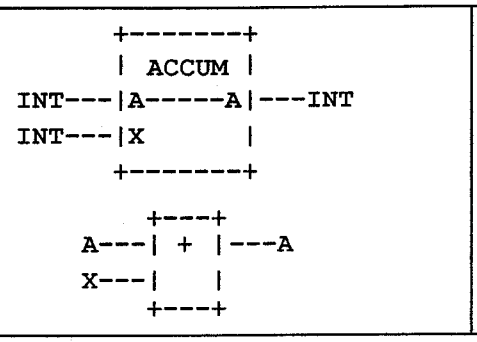
Figure 11c – Utilisation graphique d'un nom de bloc fonctionnel, en tant que variable externe (tableau 33, caractéristique N° 7b)



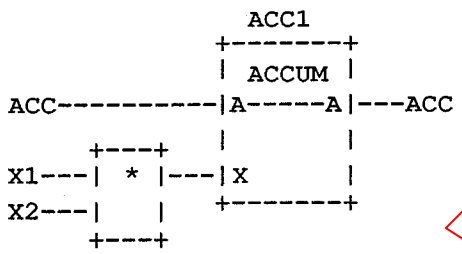
NOTE - PROGRAM declaration is defined in 2.5.3

Figure 11c - Graphical use of a function block name as an external variable (table 33, feature 7b)

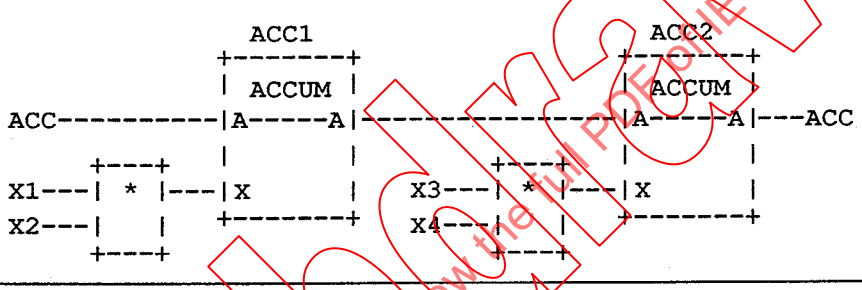
a)



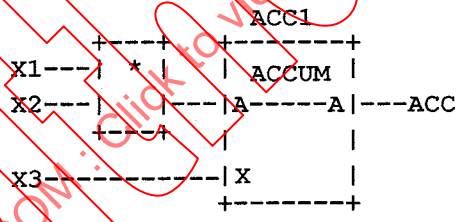
b)



c)



d)



NOTES

1 Une déclaration telle que

```
VAR
   ACC : INT ;
   X1 : INT ;
   X2 : INT ;
END_VAR
```

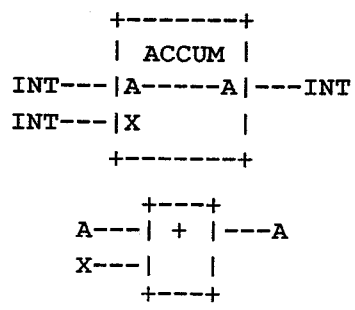
est prise implicitement.

2 Les déclarations telles que spécifiées en b) sont prises implicitement pour ACC, X1, X2, X3 et X4.

3 ILLEGAL USAGE:
L'entrée/sortie A n'est pas une variable ni un nom de bloc fonctionnel (voir texte précédent).

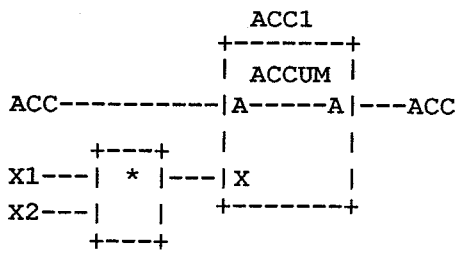
Figure 12 – Exemples d'utilisation de variables entrée/sortie
a) Déclarations graphiques et littérales
b), c) Utilisation autorisée
d) Utilisation non autorisée

a)



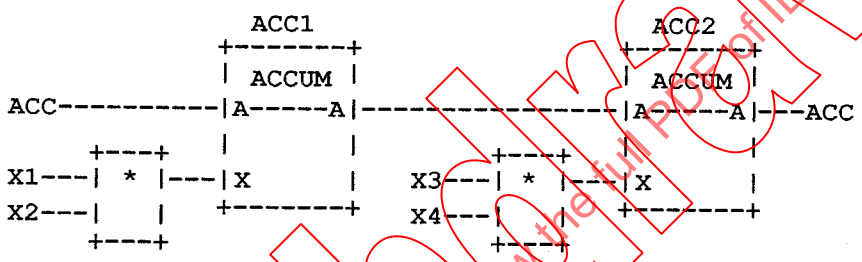

```
FUNCTION_BLOCK ACCUM
  VAR_IN_OUT A : INT ; END_VAR
  VAR_INPUT X : INT ; END_VAR
  A := A+X ;
END_FUNCTION_BLOCK
```

b)



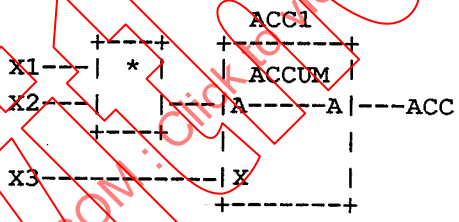
Note 1

c)



Note 2

d)



Note 3

NOTES

1 A declaration such as

```
VAR
   ACC : INT ;
   X1 : INT ;
   X2 : INT ;
END_VAR
```

is assumed.

2 Declarations as in b) are assumed for ACC, X1, X2, X3 and X4.

3 ILLEGAL USAGE:
Input/output A is not a variable or function block name (see preceding text).

Figure 12 – Examples of use of input/output variables
a) Graphical and textual declarations
b, c) Legal usage
d) Illegal usage

2.5.2.3 *Blocs fonctionnels standards*

Le présent paragraphe donne les définitions des blocs fonctionnels communs à tous les langages de programmation d'automates programmables.

Lorsque des déclarations graphiques de blocs fonctionnels standards sont indiquées dans le présent paragraphe, des déclarations littérales équivalentes, telles que spécifiées en 2.5.2.2, peuvent être également écrites, comme l'illustre, par exemple, le tableau 35.

2.5.2.3.1 *Eléments bistables*

Le tableau 34 illustre la représentation et les corps de blocs fonctionnels relatifs à des éléments bistables. La notation relative à ces éléments est choisie, de manière à être aussi cohérente que possible avec les symboles 12-09-01 et 12-09-02 de la CEI 617-12.

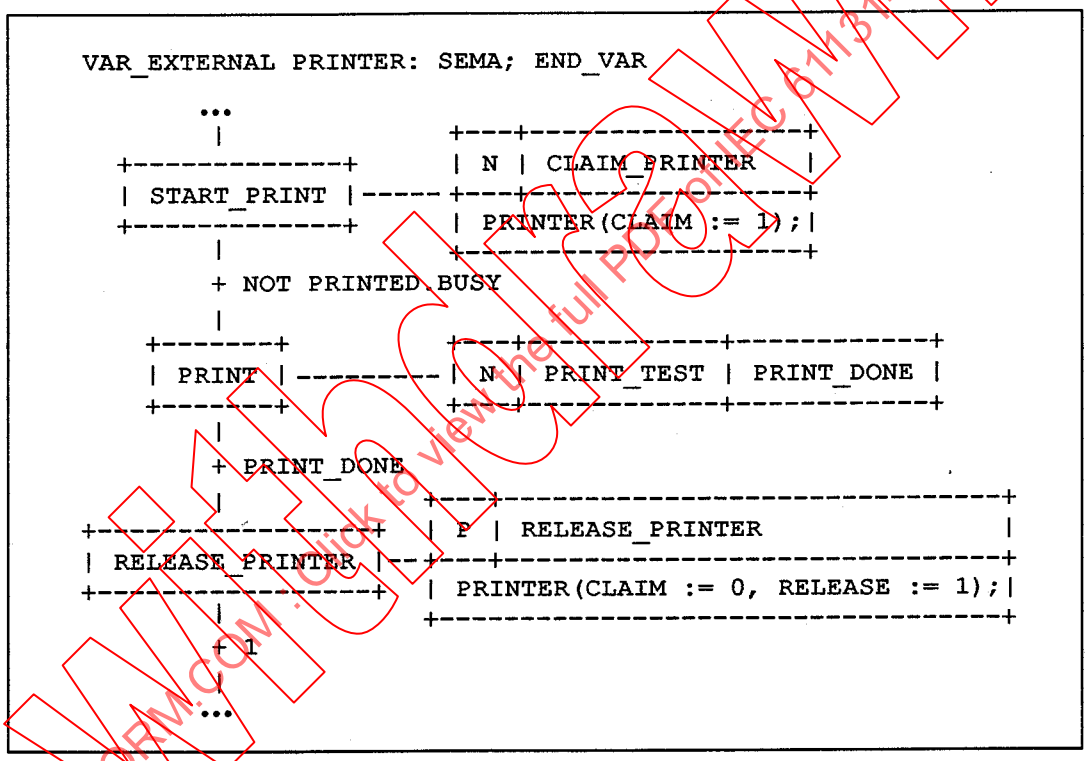


Figure 13 – Exemple d'utilisation de sémaphore (voir note 3 du tableau 34)

2.5.2.3 *Standard function blocks*

Definitions of function blocks common to all programmable controller programming languages are given in this subclause.

Where graphical declarations of standard function blocks are shown in this subclause, equivalent textual declarations, as specified in 2.5.2.2, can also be written, as for example in table 35.

2.5.2.3.1 *Bistable elements*

The representation and function block bodies for standard bistable elements are shown in table 34. The notation for these elements is chosen to be as consistent as possible with symbols 12-09-01 and 12-09-02 of IEC 617-12.

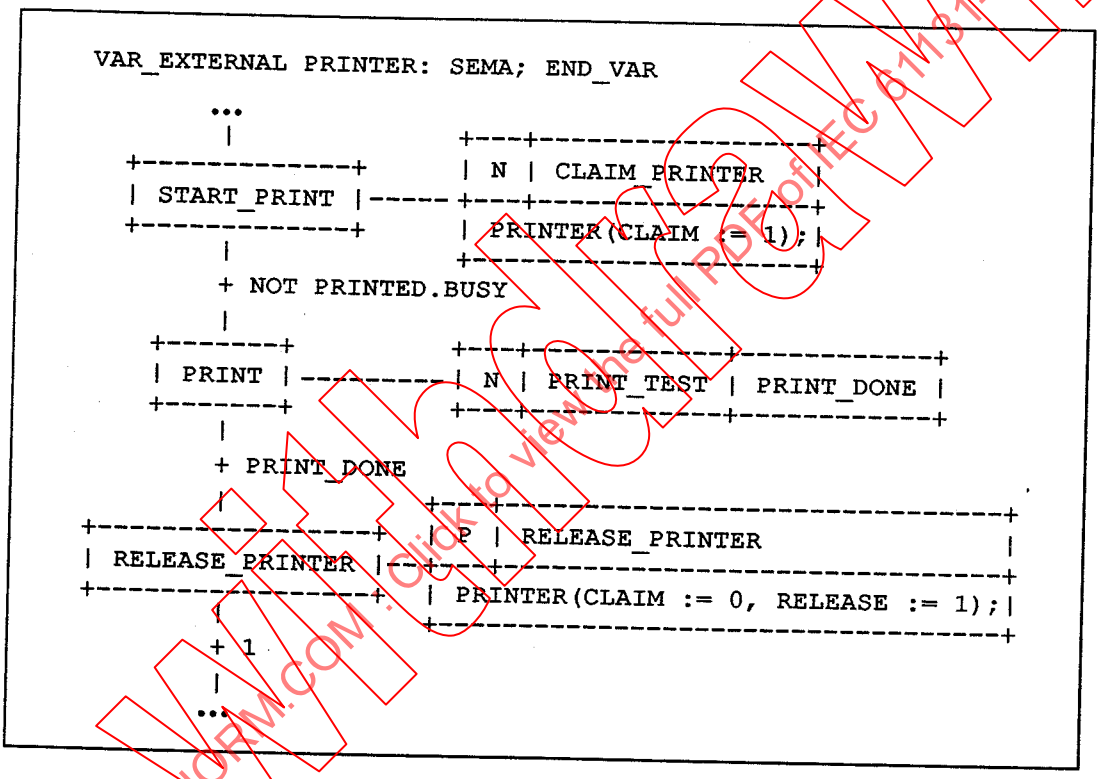
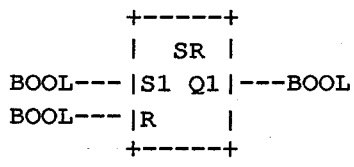
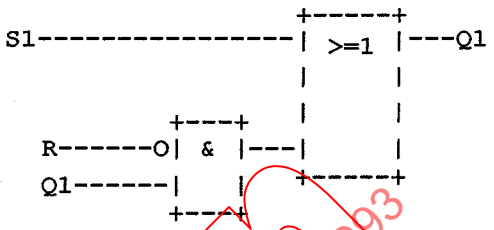
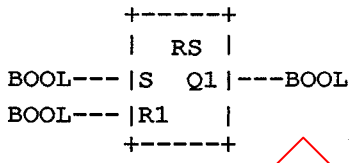
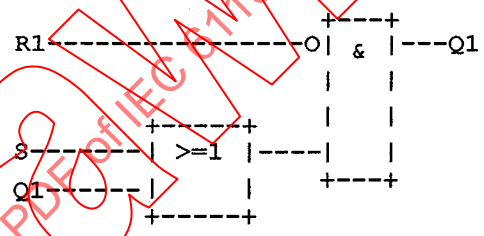
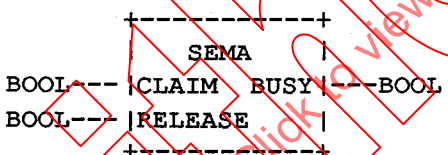


Figure 13 – Semaphore usage example (see note 4 of table 34)

Tableau 34 – Blocs fonctionnels bistables standards

<table>
<tr><th>N°</th><th>Forme graphique</th><th>Corps de bloc fonctionnel</th></tr>
<tr><td>1</td><td colspan="2">Bloc fonctionnel bistable (forcé dominant) (notes 1 et 2)</td></tr>
<tr><td></td><td> 
</td><td> 
</td></tr>
<tr><td>2</td><td colspan="2">Bloc fonctionnel bistable (réinitialisé dominant) (notes 1 et 2)</td></tr>
<tr><td></td><td> 
</td><td> 
</td></tr>
<tr><td>3</td><td colspan="2">Sémaphore avec "Essai et Positionnement" en continu (notes 3, 4, 5 et 6)</td></tr>
<tr><td></td><td> 
</td><td>VAR X : BOOL := 0 ; END_VAR
BUSY := X ;
IF CLAIM THEN X := 1 ;
ELSIF RELEASE THEN BUSY := 0 ; X := 0 ;
END_IF</td></tr>
</table>

NOTES

1 Le corps du bloc fonctionnel est spécifié dans le langage FBD défini en 4.3.

2 L'état initial de la variable de sortie Q1 doit être la valeur normale par défaut de zéro pour les variables booléennes.

3 Le corps du bloc fonctionnel est spécifié dans le langage ST défini en 3.3.

4 Ce bloc fonctionnel est destiné à être utilisé pour contrôler l'accès aux ressources du système d'exploitation; par conséquent, les deux premiers énoncés dans le corps du bloc fonctionnel, à savoir:

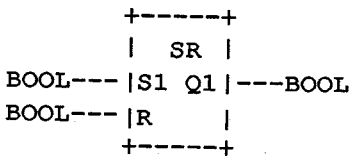
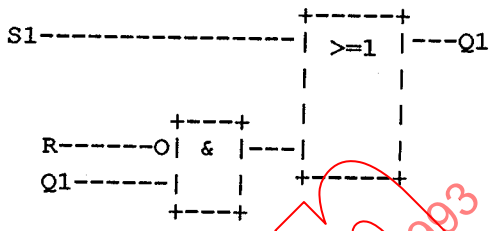
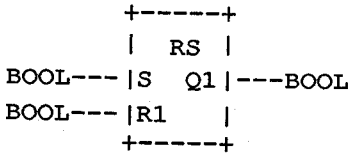
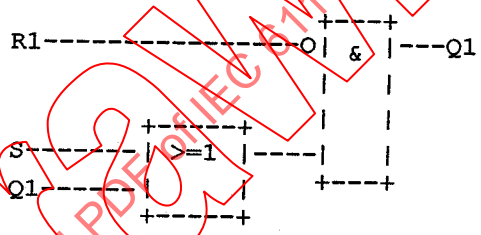
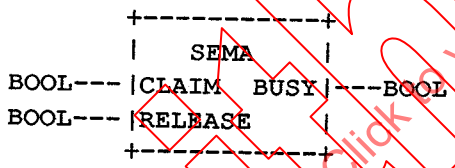
BUSY := X ; IF CLAIM THEN X := 1;

doivent être en continu.

5 Les programmes utilisateurs doivent coopérer de telle manière que seul le "détenteur" d'une sémaphore, à savoir, l'entité la plus récente pour déclencher une CLAIM sur un sémaphore inoccupé, peut libérer (RELEASE) le sémaphore.

6 La figure 13 illustre une partie de programme utilisant un sémaphore déclaré comme VAR_GLOBAL pour contrôler l'accès à la ressource imprimante, à l'aide des éléments SFC (voir 2.6).

Table 34 – Standard bistable function blocks

<table>
<tr><th>No.</th><th>Graphical form</th><th>Function block body</th></tr>
<tr><td>1</td><td colspan="2">Bistable Function Block (set dominant) (notes 1 and 2)</td></tr>
<tr><td></td><td> 
</td><td> 
</td></tr>
<tr><td>2</td><td colspan="2">Bistable Function Block (reset dominant) (notes 1 and 2)</td></tr>
<tr><td></td><td> 
</td><td> 
</td></tr>
<tr><td>3</td><td colspan="2">Semaphore with non-interruptible "Test and Set" (notes 3, 4, 5 and 6)</td></tr>
<tr><td></td><td> 
</td><td>VAR X : BOOL := 0 ; END_VAR
BUSY := X ;
IF CLAIM THEN X := 1 ;
ELSIF RELEASE THEN BUSY := 0 ; X := 0 ;
END_IF</td></tr>
</table>

NOTES

1 The function block body is specified in the Function Block Diagram (FBD) language defined in 4.3.

2 The initial state of the output variable Q1 shall be the normal default value of zero for Boolean variables.

3 The function block body is specified in the Structured Text (ST) language defined in 3.3.

4 This function block is intended to be used for controlling access to operating system resources; therefore, the first two statements in the function block body, namely,

BUSY := X ; IF CLAIM THEN X := 1;

shall be non-interruptible.

5 User programs must co-operate in such a way that only the "owner" of a semaphore, that is, the most recent entity to successfully assert a CLAIM on a non-BUSY semaphore, can RELEASE the semaphore.

6 Figure 13 shows a program fragment using a semaphore declared as VAR_GLOBAL to control access to a printer resource, using SFC elements (see 2.6).

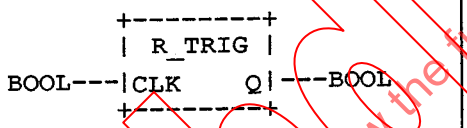
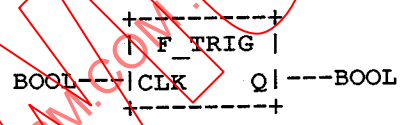
2.5.2.3.2 *Détection de fronts*

La représentation graphique de blocs fonctionnels de détection de fronts montants et de fronts descendants doit être conforme aux indications du tableau 35. Le comportement de ces blocs doit être équivalent aux définitions données dans ce tableau. Ce comportement correspond aux règles suivantes:

1) La sortie "Q" d'un bloc fonctionnel R_TRIG doit rester à la valeur booléenne "1", d'une exécution du bloc fonctionnel à la suivante, suivant la transition de "0" à "1" de l'entrée "CLK", et doit retourner à "0", lors de l'exécution suivante.

2) La sortie "Q" d'un bloc fonctionnel F_TRIG doit rester à la valeur booléenne "1", d'une exécution du bloc fonctionnel à la suivante, suivant la transition de "1" à "0" de l'entrée "CLK", et doit retourner à "0", lors de l'exécution suivante.

Tableau 35 – Blocs fonctionnels standards de détection de fronts

N°	Forme graphique	Définition (langage ST – voir 3.3)
1	Détecteur de front montant	
		FUNCTION_BLOCK R_TRIG VAR_INPUT CLK : BOOL ; END_VAR VAR_OUTPUT Q : BOOL ; END_VAR VAR M : BOOL := 0 ; END_VAR Q := CLK AND NOT M ; M := CLK ; END_FUNCTION_BLOCK
2	Détecteur de front descendant	
		FUNCTION_BLOCK F_TRIG VAR_INPUT CLK : BOOL ; END_VAR VAR_OUTPUT Q : BOOL ; END_VAR VAR M : BOOL := 1 ; END_VAR Q := NOT CLK AND NOT M ; M := NOT CLK ; END_FUNCTION_BLOCK

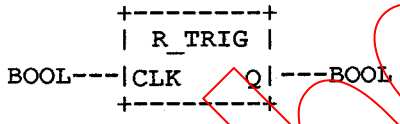
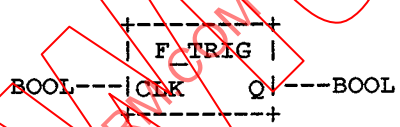
2.5.2.3.2 *Edge detection*

The graphic representation of standard rising- and falling-edge detecting function blocks shall be as shown in table 35. The behaviors of these blocks shall be equivalent to the definitions given in this table. This behavior corresponds to the following rules:

1) The "Q" output of an R_TRIG function block shall stand at the Boolean "1" value from one execution of the function block to the next, following the "0" to "1" transition of the "CLK" input, and shall return to "0" at the next execution.

2) The "Q" output of an F_TRIG function block shall stand at the Boolean "1" value from one execution of the function block to the next, following the "1" to "0" transition of the "CLK" input, and shall return to "0" at the next execution.

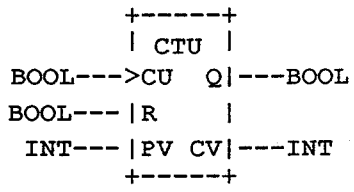
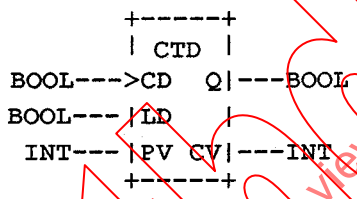
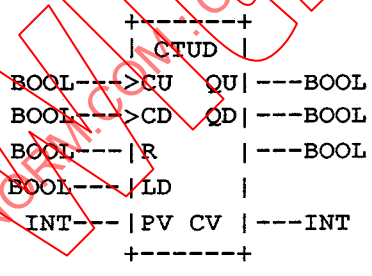
Table 35 – Standard edge detection function blocks

No.	Graphical form	Definition (ST language – see 3.3)
1	Rising edge detector	
		FUNCTION_BLOCK R_TRIG VAR_INPUT CLK : BOOL ; END_VAR VAR_OUTPUT Q : BOOL ; END_VAR VAR M : BOOL := 0 ; END_VAR Q := CLK AND NOT M ; M := CLK ; END_FUNCTION_BLOCK
2	Falling edge detector	
		FUNCTION_BLOCK F_TRIG VAR_INPUT CLK : BOOL ; END_VAR VAR_OUTPUT Q : BOOL ; END_VAR VAR M : BOOL := 1 ; END_VAR Q := NOT CLK AND NOT M ; M := NOT CLK ; END_FUNCTION_BLOCK

2.5.2.3.3 *Compteurs*

Le tableau 36 illustre les représentations graphiques de blocs fonctionnels standards de compteurs, accompagnées des types d'entrées et de sorties associées. Le fonctionnement de ces blocs fonctionnels doit être tel que spécifié dans les corps des blocs fonctionnels correspondants.

Tableau 36 – Blocs fonctionnels standards de compteurs

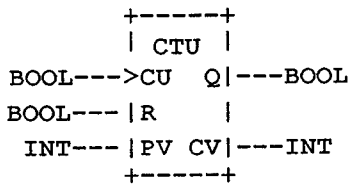
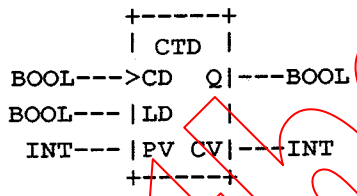
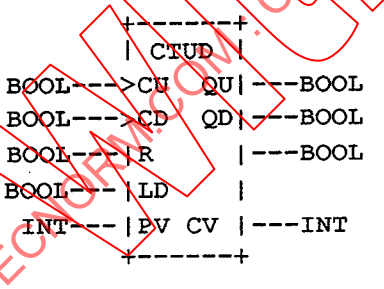
N°	Forme graphique	Corps de bloc fonctionnel (langage ST – voir 3.3)
1	Compteur (comptage)	
		IF R THEN CV := 0 ; ELSIF CU AND (CV < PVmax) THEN CV := CV+1 ; END_IF ; Q := (CV >= PV) ;
2	Compteur (décomptage)	
		IF LD THEN CV := PV ; ELSIF CD AND (CV > PVmin) THEN CV := CV-1 ; END_IF ; Q := (CV <= 0) ;
3	Compteur (comptage-décomptage)	
		IF R THEN CV := 0 ; ELSIF LD THEN CV := PV ; ELSIF CU AND (CV < PVmax) THEN CV := CV+1 ; ELSIF CD AND (CV > PVmin) THEN CV := CV-1 ; END_IF ; QU := (CV >= PV) ; QD := (CV <= 0) ;

NOTE - Les valeurs numériques des variables limites PVmin et PVmax sont propres à l'application concernée.

2.5.2.3.3 *Counters*

The graphic representations of standard counter function blocks, with the types of the associated inputs and outputs, shall be as shown in table 36. The operation of these function blocks shall be as specified in the corresponding function block bodies.

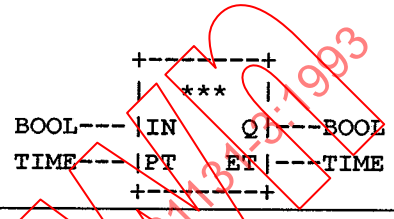
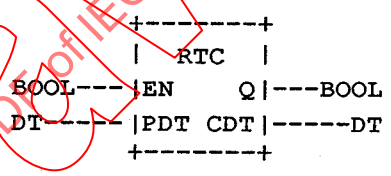
Table 36 – Standard counter function blocks

No.	Graphical form	Function block body (ST language – see 3.3)
1	Up-counter	
		IF R THEN CV := 0 ; ELSIF CU AND (CV < PVmax) THEN CV := CV+1 ; END_IF ; Q := (CV >= PV) ;
2	Down-counter	
		IF LD THEN CV := PV ; ELSIF CD AND (CV > PVmin) THEN CV := CV-1 ; END_IF ; Q := (CV <= 0) ;
3	Up-down counter	
		IF R THEN CV := 0 ; ELSIF LD THEN CV := PV ; ELSIF CU AND (CV < PVmax) THEN CV := CV+1 ; ELSIF CD AND (CV > PVmin) THEN CV := CV-1 ; END_IF ; QU := (CV >= PV) ; QD := (CV <= 0) ;
NOTE - The numerical values of the limit variables PVmin and PVmax are implementation-dependent.		

2.5.2.3.4 *Temporisateurs*

La forme graphique pour des blocs fonctionnels standards de temporisateurs doit être telle qu'illustrée au tableau 37. Le fonctionnement de ces blocs fonctionnels doit être tel que défini dans les schémas de temporisation indiqués au tableau 38.

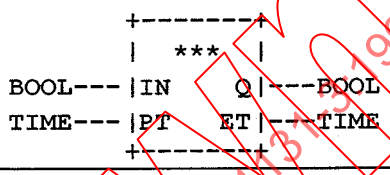
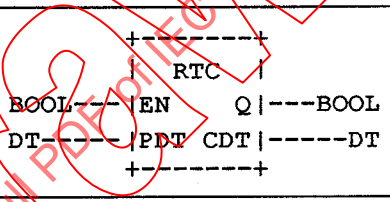
Tableau 37 – Blocs fonctionnels standards de temporisateurs

<table>
<tr><th>N°</th><th>Description</th><th>Forme graphique</th></tr>
<tr><td>1</td><td>***est: TP (impulsion)</td><td rowspan="4">  </td></tr>
<tr><td>2a</td><td>TON (Enclenchement)</td></tr>
<tr><td>2b</td><td>T---0 (Enclenchement)</td></tr>
<tr><td>3a</td><td>TOF (Déclenchement)</td></tr>
<tr><td>3b</td><td>0---T (Déclenchement)</td><td></td></tr>
<tr><td>4</td><td colspan="2">Horloge temps réel</td></tr>
<tr><td></td><td>PDT = Date et heure prédéfinies, chargées sur le front montant de EN
CDT = Date et heure du jour, valables lorsque EN=1
Q = copie de EN</td><td>  </td></tr>
<tr><td colspan="3">NOTE - Dans les langages littéraux, les caractéristiques 2b et 3b ne doivent pas être utilisées.</td></tr>
</table>

2.5.2.3.4 *Timers*

The graphic form for standard timer function blocks shall be as shown in table 37. The operation of these function blocks shall be as defined in the timing diagrams given in table 38.

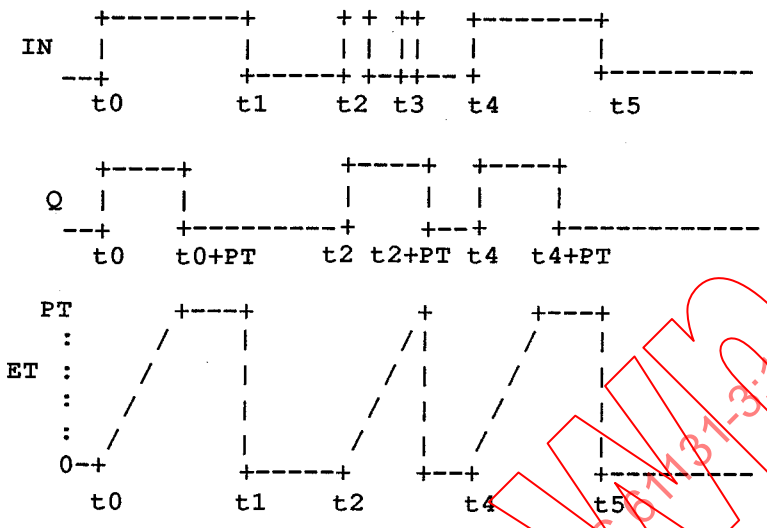
Table 37 – Standard timer function blocks

No.	Description	Graphic form
1	***is: TP (Pulse)	
2a	TON (On-delay)	
2b	T---0 (On-delay)	
3a	TOF (Off-delay)	
3b	0---T (Off-delay)	
4	Real-time clock	
	PDT = Preset date and time, loaded on rising edge of EN CDT = Current date and time, valid when EN=1 Q = copy of EN	

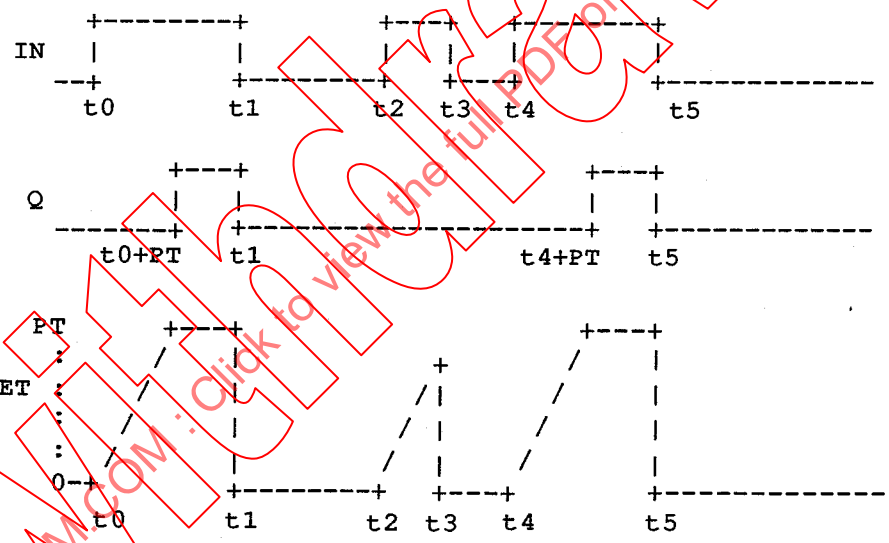
NOTE - In textual languages, features 2b and 3b shall not be used.

Tableau 38 – Blocs fonctionnels standards de temporisateurs - schémas de temporisation

Temporisation d'impulsion (TP)



Temporisation d'enclenchement (TON)



Temporisation de déclenchement (TOF)

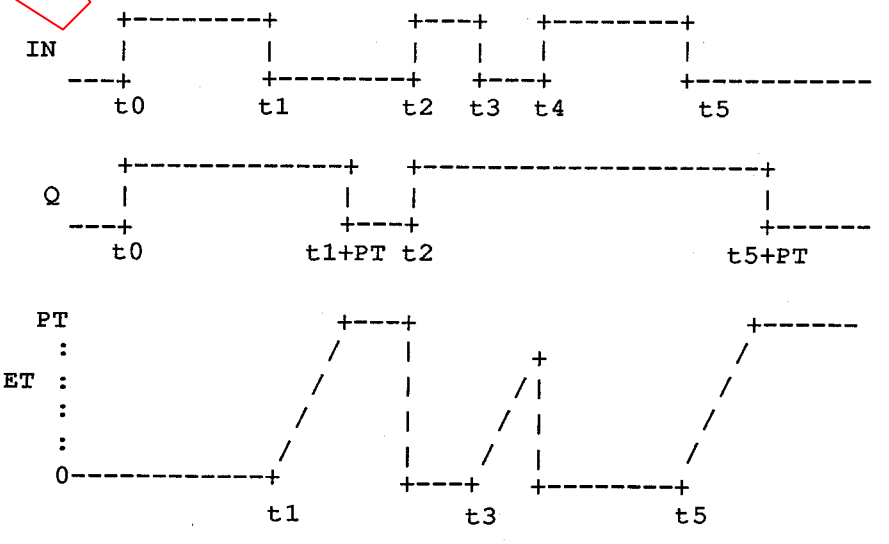
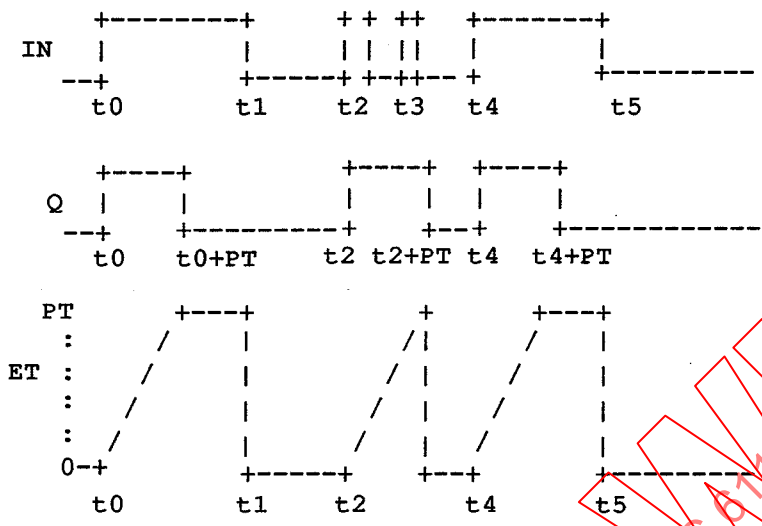
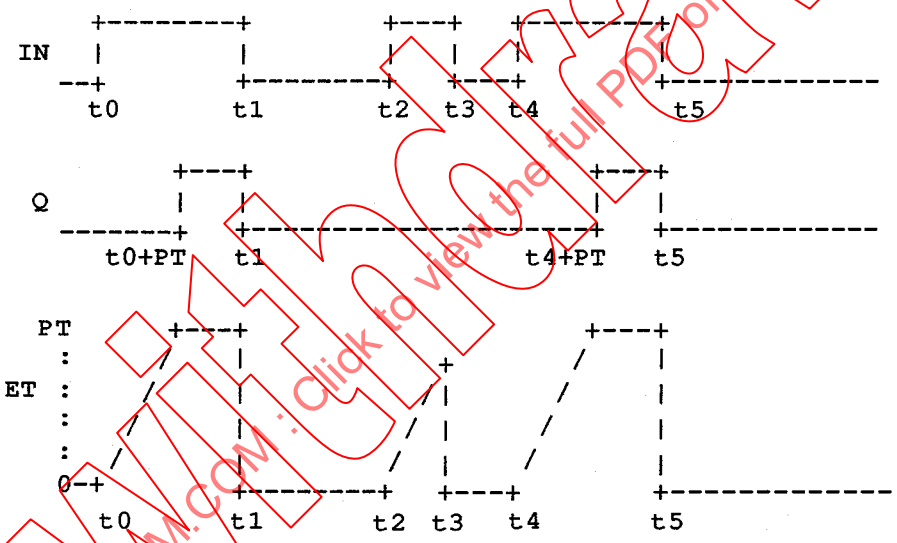


Table 38 – Standard timer function blocks – timing diagrams

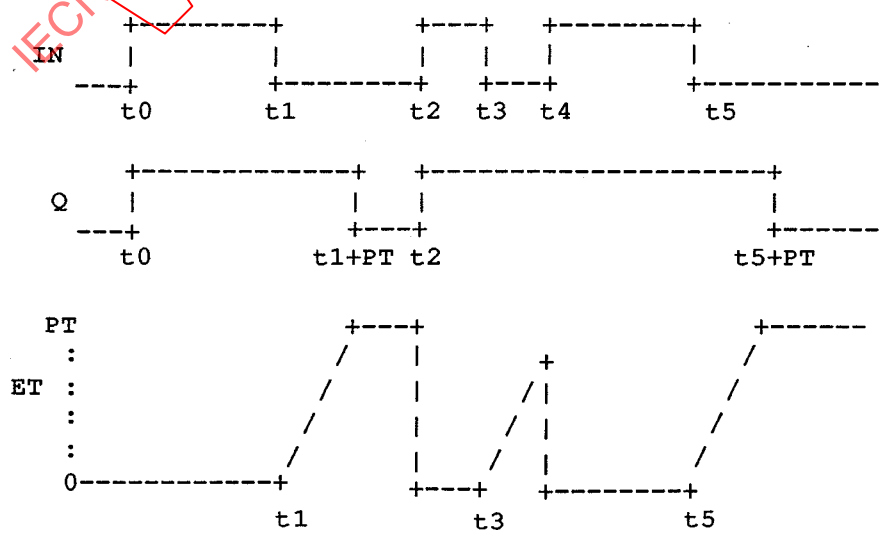
Pulse (TP) timing



On-delay (TON) timing



Off-delay (TOF) timing



2.5.2.3.5 *Blocs fonctionnels de communication*

Les blocs fonctionnels standards de communication, relatifs aux automates programmables, sont définis dans la CEI 1131-5. Ces blocs fonctionnels assurent une fonctionnalité de transmissions programmable, telle que la vérification de dispositifs, l'acquisition de données appelables, l'acquisition de données programmées, la commande paramétrique, la commande verrouillée, l'indication d'alarmes programmées, ainsi que la gestion et la protection des liaisons.

2.5.3 *Programmes*

Un *programme* est défini, dans la CEI 1131-1, comme un "ensemble logique de tous les éléments et constructions des langages de programmation nécessaires pour le traitement des signaux destinés à commander une machine ou au moyen d'une configuration d'automate programmable".

Le paragraphe 1.4.1 de la présente partie décrit la position des programmes dans le modèle logiciel global d'un automate programmable; le paragraphe 1.4.2 décrit les moyens disponibles pour la communication interne à un programme ou pour la communication entre programmes; le paragraphe 1.4.3 décrit le processus général de développement d'un programme.

La déclaration et l'utilisation de *programmes* sont identiques à celles des *blocs fonctionnels* tels que définis en 2.5.2.1 et 2.5.2.2, avec les caractéristiques supplémentaires indiquées au tableau 39 et les différences suivantes:

1) Les mots clés de séparation des déclarations de programmes doivent être: PROGRAM...END_PROGRAM.

2) Un programme peut contenir une construction VAR_ACCESS...VAR_END qui fournit un moyen de spécifier des variables nommées auxquelles on peut avoir accès au moyen de certains des services de communication spécifiés dans la CEI 1131-5. Un chemin d'accès associe chacune de ces variables à une entrée, sortie ou variable interne d'un *programme*. Le format et l'utilisation de cette déclaration doivent être tels que décrits en 2.7.1 et dans la CEI 1131-5.

3) Des *programmes* ne peuvent être instanciés que dans des *ressources,* telles que définies en 2.7.1, alors que des *blocs fonctionnels* ne peuvent être instanciés que dans des *programmes* ou dans d'autres *blocs fonctionnels.*

La déclaration et l'utilisation de programmes sont illustrées dans la figure 19, ainsi que dans les exemples F.7 et F.8 de l'annexe F.

Tableau 39 – Caractéristiques de déclarations de programmes

N°	Description
1 à 9b	Identiques aux caractéristiques 1 à 9b du tableau 33
10	Paramètres formels d'entrée et de sortie
11 à 14	Respectivement identiques aux caractéristiques 1 à 4 du tableau 17
15 à 18	Respectivement identiques aux caractéristiques 1 à 4 du tableau 18
19	Utilisation de variables directement représentées (voir 2.4.1.1)
20	Déclaration VAR_GLOBAL ... END_VAR dans un PROGRAM (voir 2.4.3 et 2.7.1)
21	Déclaration VAR_ACCESS .. END_VAR dans un PROGRAM

2.5.2.3.5 *Communication function blocks*

Standard communication function blocks for programmable controllers are defined in IEC 1131-5. These function blocks provide programmable communications functionality such as device verification, polled data acquisition, programmed data acquisition, parametric control, interlocked control, programmed alarm reporting, and connection management and protection.

2.5.3 *Programs*

A *program* is defined in IEC 1131-1 as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system."

Subclause 1.4.1 of this part describes the place of programs in the overall software model of a programmable controller; subclause 1.4.2 describes the means available for inter- and intra-program communication; and subclause 1.4.3 describes the overall process of program development.

The declaration and usage of *programs* is identical to that of *function blocks* as defined in 2.5.2.1 and 2.5.2.2, with the additional features shown in table 39 and the following differences:

1) The delimiting keywords for program declarations shall be PROGRAM...END_PROGRAM.

2) A program can contain a VAR_ACCESS...END_VAR construction, which provides a means of specifying named variables which can be accessed by some of the communication services specified in IEC 1131-5. An *access path* associates each such variable with an input, output or internal variable of the program. The format and usage of this declaration shall be as described in 2.7.1 and in IEC 1131-5.

3) *Programs* can only be instantiated within *resources*, as defined in 2.7.1, while *function blocks* can only be instantiated within *programs* or other *function blocks*.

The declaration and use of programs are illustrated in figure 19, and in examples F.7 and F.8 of annex F.

Table 39 – Program declaration features

No.	Description
1 to 9b	Same as features 1 to 9b, respectively, of table 33
10	Formal input and output parameters
11 to 14	Same as features 1 to 4, respectively, of table 17
15 to 18	Same as features 1 to 4, respectively, of table 18
19	Use of directly represented variables (see 2.4.1.1)
20	VAR_GLOBAL ... END_VAR declaration within a PROGRAM (see 2.4.3 and 2.7.1)
21	VAR_ACCESS .. END_VAR declaration within a PROGRAM

2.6 *Eléments du diagramme fonctionnel en séquence (SFC)*

2.6.1 *Généralités*

Le présent paragraphe définit les éléments du *diagramme fonctionnel en séquence* (SFC), destinés à être utilisés pour la structuration de l'organisation interne d'une unité d'organisation de programme d'automate programmable, écrits dans l'un des langages définis dans la présente norme, dans le but d'effectuer des fonctions de *commandes séquentielles.* Les définitions données dans le présent paragraphe sont celles de la CEI 848, avec les modifications nécessaires pour convertir les représentations prises, provenant d'une *documentation normative,* en un ensemble d'*éléments de commande d'exécution* relatif à une unité d'organisation de programme d'automate programmable.

Les éléments du diagramme fonctionnel en séquence fournissent un moyen permettant de segmenter une unité d'organisation de programme d'automate programmable en un ensemble d'*étapes* et de *transitions* reliées entre elles par des *liaisons dirigées.* A chaque étape est associé un ensemble d'actions, et à chaque transition est associée une *condition de transition.*

Dans la mesure où les éléments du diagramme fonctionnel en séquence nécessitent une mémorisation des informations relatives à l'état, les seules unités d'organisation de programmes qui peuvent être structurées à l'aide de ces éléments sont les *blocs fonctionnels* et des *programmes.*

Si une partie d'unité d'organisation de programme est segmentée en éléments de diagramme fonctionnel en séquence, l'intégralité de l'unité d'organisation de programme doit être également segmentée. Si aucune segmentation du diagramme fonctionnel en séquence n'est donnée pour une unité d'organisation de programme, l'intégralité de l'unité d'organisation de programme doit être considérée comme une action exécutée sous le contrôle de l'entité lancée.

2.6.2 *Etapes*

Une *étape* représente une situation dans laquelle le comportement d'une unité d'organisation de programme, par rapport à ses entrées et à ses sorties, suit un ensemble de règles définies par les actions associées de l'étape. Une étape est soit *active,* soit *inactive;* à tout moment, l'état de l'unité d'organisation de programme est défini par l'ensemble d'étapes actives ainsi que par les valeurs de ses variables internes et de ses valeurs de sortie.

Comme l'illustre le tableau 40, une étape doit être représentée graphiquement par un bloc contenant un *nom d'étape* sous la forme d'un identificateur tel que défini en 2.1.2, ou littéralement par une construction STEP...END_STEP. Dans l'étape, la ou les liaisons dirigées peuvent être représentées graphiquement par une ligne verticale rattachée à la partie supérieure de l'étape. La ou les liaisons dirigées, situées en dehors de l'étape, peuvent être représentées par une ligne verticale rattachée à la partie inférieure de l'étape. Alternativement, les liaisons dirigées peuvent être représentées littéralement par la construction TRANSITION...END_TRANSITION définie en 2.6.3.

2.6 *Sequential Function Chart (SFC) elements*

2.6.1 *General*

This subclause defines *sequential function chart* (SFC) elements for use in structuring the internal organization of a programmable controller program organization unit, written in one of the languages defined in this standard, for the purpose of performing *sequential control* functions. The definitions in this subclause are derived from IEC 848, with the changes necessary to convert the representations from a *documentation standard* to a set of *execution control elements* for a programmable controller program organization unit.

The SFC elements provide a means of partitioning a programmable controller program organization unit into a set of *steps* and *transitions* interconnected by *directed links*. Associated with each step is a set of *actions*, and with each transition is associated a *transition condition*.

Since SFC elements require storage of state information, the only program organization units which can be structured using these elements are *function blocks* and *programs*.

If any part of a program organization unit is partitioned into SFC elements, the entire program organization unit shall be so partitioned. If no SFC partitioning is given for a program organization unit, the entire program organization unit shall be considered to be a single *action* which executes under the control of the invoking entity.

2.6.2 *Steps*

A *step* represents a situation in which the behavior of a program organization unit with respect to its inputs and outputs follows a set of rules defined by the associated *actions* of the step. A step is either *active* or *inactive*. At any given moment, the state of the program organization unit is defined by the set of active steps and the values of its internal and output variables.

As shown in table 40, a step shall be represented graphically by a block containing a *step name* in the form of an identifier as defined in 2.1.2, or textually by a STEP...END_STEP construction. The directed link(s) into the step can be represented graphically by a vertical line attached to the top of the step. The directed link(s) out of the step can be represented by a vertical line attached to the bottom of the step. Alternatively, the directed links can be represented textually by the TRANSITION...END_TRANSITION construction defined in 2.6.3.

Le *drapeau d'étape* (état actif ou inactif d'une étape) peut être représenté par la valeur logique d'un élément de structure booléen ***.X, où *** est le nom de l'étape, comme cela est indiqué au tableau 40. Cette variable booléenne a la valeur "1" lorsque l'étape correspondante est active, et "0" lorsqu'elle est inactive. L'état de cette variable est disponible pour la connexion graphique au côté droit de l'étape, comme l'indique le tableau 40.

De façon similaire, le temps écoulé, ***.T, depuis l'initialisation d'une étape peut être représenté par un élément de structure de type TIME (temps), tel qu'indiqué au tableau 40. Lorsqu'une étape est désactivée, la valeur du temps écoulé pour l'étape doit conserver la valeur qu'il avait au moment où l'étape a été désactivée. Lorsqu'une étape est activée, la valeur du temps écoulé pour l'étape doit être remis à t#0s.

Le *domaine* de noms d'étapes, de drapeaux d'étapes et de temps d'étapes, doit être *local* à l'unité d'organisation de programme dans laquelle les étapes apparaissent.

L'état initial de l'unité d'organisation de programme est représenté par les valeurs initiales de ses variables internes et de ses variables de sortie, et par son jeu d'*étapes initiales,* à savoir, les étapes qui sont initialement actives. Chaque *réseau* de diagramme fonctionnel en séquence (SFC), ou son équivalent littéral, doit avoir exactement une étape initiale.

Une étape initiale peut être représentée graphiquement par des lignes doubles pour les bordures; avec le jeu de caractères ISO/IEC 646, elle doit être représentée conformément au tableau 40. Une étape initiale peut être représentée littéralement par la construction INITIAL_STEP...END_STEP indiquée au tableau 40.

En ce qui concerne l'initialisation, telle que définie en 2.4.2, le temps initial écoulé par défaut, relatif aux étapes, est t#0s, et l'état initial par défaut est le 0 booléen, relatif aux étapes ordinaires et le 1 booléen pour les étapes initiales. Cependant, lorsqu'une instance de bloc fonctionnel ou de programme est déclarée comme *non volatile,* (comme, par exemple, dans la caractéristique 3 du tableau 33), les états et les temps écoulés (s'ils sont acceptés) de toutes les étapes contenues dans le programme ou dans le bloc fonctionnel doivent être traités comme des états et des temps écoulés non volatiles pour l'initialisation du système, telle que définie en 2.4.2.

The *step flag* (active or inactive state of a step) can be represented by the logic value of a Boolean structure element ***.X, where *** is the step name, as shown in table 40. This Boolean variable has the value "1" when the corresponding step is active, and "0" when it is inactive. The state of this variable is available for graphical connection at the right side of the step as shown in table 40.

Similarly, the elapsed time, ***.T, since initiation of a step can be represented by a structure element of type TIME, as shown in table 40. When a step is deactivated, the value of the step elapsed time shall remain at the value it had when the step was deactivated. When a step is activated, the value of the step elapsed time shall be reset to t#0s.

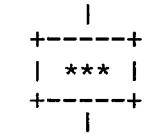
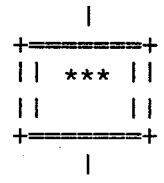
The *scope* of step names, step flags, and step times shall be *local* to the program organization unit in which the steps appear.

The initial state of the program organization unit is represented by the initial values of its internal and output variables, and by its set of *initial steps*, i.e., the steps which are initially active. Each SFC *network*, or its textual equivalent, shall have exactly one initial step.

An initial step can be drawn graphically with double lines for the borders, and with the ISO/IEC 646 character set shall be drawn as shown in table 40.

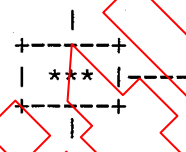
For system initialization as defined in 2.4.2, the default initial elapsed time for steps is t#0s, and the default initial state is Boolean 0 for ordinary steps and Boolean 1 for initial steps. However, when an instance of a function block or a program is declared to be *retentive* (for instance, as in feature 3 of table 33), the states and (if supported) elapsed times of all steps contained in the program or function block shall be treated as retentive for system initialization as defined in 2.4.2.

Tableau 40 – Caractéristiques d'étape

N°	Représentation	Description
1		Etape - Forme graphique avec liaisons dirigées "***" = Nom d'étape
		Etape initiale - Forme graphique avec liaisons dirigées "***" = Nom d'étape initiale (note 2)

2	`STEP *** :`
	`(* Corps d'étape *)`
	`END_STEP`
		Etape - Forme littérale sans liaisons dirigées (voir 2.6.3)
		"***" = Nom d'étape

	`INITIAL_STEP *** :`
	`(* Corps d'étape *)`
	`END_STEP`
		Etape initiale - Forme littérale sans liaisons dirigées (voir 2.6.3)
		"***" = Nom d'étape

3a	`***.X`	Drapeau d'étape - Forme générale "***" = Nom d'étape ***.X = 1 booléen quand *** est actif = sinon 0 booléen
3b		Drapeau d'étape - Connexion directe de variable booléenne ***.X à droite de l'étape "***"
4	`***.T`	Temps écoulé pour une étape - Forme générale "***" = Nom d'étape ***.T = une variable de type TIME (voir définition 2.6.2)

NOTES

1 Lorsque la caractéristique 3a, 3b, ou 4 est acceptée, elle doit constituer une erreur si le programme utilisateur tente de modifier la variable associée. Par exemple: si S4 est un nom d'étape, alors les énoncés suivants devraient être des erreurs dans le langage ST défini en 3.3:

S4.X := 1 ; (* ERROR *)

S4.T := t#100ms ; (* ERROR *)

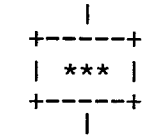
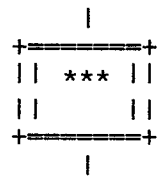
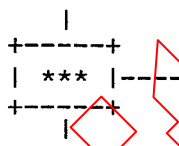
2 La liaison dirigée supérieure n'est pas requise si l'étape initiale n'a pas de prédécesseur.

2.6.3 *Transitions*

Une *transition* représente la condition par laquelle la commande passe d'une ou plusieurs étapes précédant la transition à une ou plusieurs étapes succédant à la transition le long de la liaison dirigée correspondante. La transition doit être représentée par une ligne horizontale en travers de la liaison dirigée verticale.

La direction de l'évolution suivant les liaisons dirigées doit partir de la partie inférieure de l'étape ou des étapes précédentes et aboutir à la partie supérieure de l'étape ou des étapes suivantes.

Table 40 – Step features

No.	Representation	Description
1		Step - Graphical form with directed links "***" = step name
		Initial step - Graphical form with directed links "***" = Name of initial step (note 2)
2	STEP *** : (* Step body *) END_STEP	Step - Textual form without directed links (see 2.6.3) "***" = Step name
	INITIAL_STEP *** : (* Step body *) END_STEP	Initial step - Textual form without directed links (see 2.6.3) "***" = Name of initial step
3a	***.X	Step flag - General form "***" = Step name ***.X = Boolean 1 when *** is active, Boolean 0 otherwise
3b		Step flag - Direct connection of Boolean variable ***.X to right side of step "***"
4	***.T	Step elapsed time - General form "***" = Step name ***.T = A variable of type TIME (see 2.6.2)

NOTES

1 When feature 3a, 3b, or 4 is supported, it shall be an *error* if the user program attempts to modify the associated variable. For example, if S4 is a step name, then the following statements would be *errors* in the ST language defined in 3.3:

S4.X := 1 ; (* ERROR *)

S4.T := t#100ms ; (* ERROR *)

2 The upper directed link is not required if the initial step has no predecessors.

2.6.3 *Transition*

A *transition* represents the condition whereby control passes from one or more steps preceding the transition to one or more successor steps along the corresponding directed link. The transition shall be represented by a horizontal line across the vertical directed link.

The direction of evolution following the directed links shall be from the bottom of the predecessor step(s) to the top of the successor step(s).

A chaque transition doit être associée une *condition de transition* qui est le résultat de l'évaluation d'une expression booléenne unique. Une condition de transition qui est toujours vraie doit être représentée par le symbole "1" ou par le mot clé TRUE (vrai).

Une condition de transition peut être associée à une transition par l'un des moyens suivants, comme l'indique le tableau 41:

1) En plaçant l'expression booléenne appropriée, exprimée dans le langage ST défini en 3.3, à droite de la liaison dirigée verticale.

2) Par un réseau de schémas à contacts, exprimé dans le langage LD défini en 4.2, dont la sortie coupe la liaison dirigée verticale au lieu d'une barre droite.

3) Par un réseau, exprimé dans le langage FBD défini en 4.3, dont la sortie coupe la liaison dirigée verticale.

4) Par un réseau LD ou FBD dont la sortie coupe la liaison dirigée verticale par l'intermédiaire d'un *connecteur* tel que défini en 4.1.1.

5) Par une construction TRANSITION...END_TRANSITION utilisant le langage ST. Celle-ci doit se composer de ce qui suit:

- du mot clé TRANSITION FROM, suivi du nom d'étape de l'étape précédente (ou, s'il y a plus d'une étape précédente, suivi d'une liste d'étapes précédentes, mise entre parenthèses);
- le mot clé TO suivi du nom d'étape de l'étape suivante (ou, s'il y a plus d'une étape suivante, suivi d'une liste d'étapes suivantes, mise entre parenthèses);
- l'opérateur d'affectation (:=), suivi d'une expression booléenne en langage ST, spécifiant la condition de transition;
- le mot clé de fin END_TRANSITION.

6) Par une construction TRANSITION...END_TRANSITION utilisant le langage IL défini en 3.2. Cette construction doit se composer de ce qui suit:

- les mots clés TRANSITION FROM, suivi du nom d'étape de l'étape précédente (ou, s'il y a plus d'une étape précédente, suivi d'une liste d'étapes précédentes, mise entre parenthèses);
- le mot clé TO, suivi du nom d'étape de l'étape suivante (ou, s'il y a plus d'une étape suivante, suivi d'une liste d'étapes suivantes, mise entre parenthèses) suivi d'un deux points (':');
- commençant par une ligne séparée, une liste d'instructions en langage IL, dont le résultat de l'évaluation détermine la condition de transition;
- le mot clé de fin END_TRANSITION, sur une ligne séparée.

7) En utilisant un *nom de transition* sous la forme d'un identificateur placé à droite de la liaison dirigée. Cet identificateur doit se rapporter à une construction TRANSITION...END_TRANSITION définissant une des entités suivantes, dont l'évaluation doit aboutir à l'affectation d'une valeur booléenne à la variable désignée par le nom de transition:

- un réseau en langage LD ou FBD;
- une liste d'instructions en langage IL;
- une affectation d'expression booléenne en langage ST.

Le *domaine* d'un nom de transition doit être *local* à l'unité d'organisation de programme dans laquelle se trouve la transition.

Elle doit constituer une *erreur* au sens de 1.5.1, si un éventuel "effet indésiré" (par exemple, l'affectation d'une valeur à une variable autre que le nom de transition) se produit pendant l'évaluation d'une condition de transition.

Each transition shall have an associated *transition condition* which is the result of the evaluation of a single Boolean expression. A transition condition which is always true shall be represented by the symbol "1" or the keyword TRUE.

A transition condition can be associated with a transition by one of the following means, as shown in table 41:

1) By placing the appropriate Boolean expression in the ST language defined in 3.3 to the right of the vertical directed link.

2) By a ladder diagram network in the LD language defined in 4.2, whose output intersects the vertical directed link instead of a right rail.

3) By a network in the FBD language defined in 4.3, whose output intersects the vertical directed link.

4) By a LD or FBD network whose output intersects the vertical directed link via a *connector* as defined in 4.1.1.

5) By a TRANSITION...END_TRANSITION construct using the ST language. This shall consist of:

- The keywords TRANSITION FROM followed by the step name of the predecessor step (or, if there is more than one predecessor, by a parenthesized list of predecessor steps);
- The keyword TO followed by the step name of the successor step (or, if there is more than one successor, by a parenthesized list of successor steps);
- The assignment operator (:=), followed by a Boolean expression in the ST language, specifying the transition condition;
- The terminating keyword END_TRANSITION.

6) By a TRANSITION...END_TRANSITION construct using the IL language defined in 3.2. This shall consist of:

- The keywords TRANSITION FROM followed by the step name of the predecessor step (or, if there is more than one predecessor, by a parenthesized list of predecessor steps), followed by a colon (':');
- The keyword TO followed by the step name of the successor step (or, if there is more than one successor, by a parenthesized list of successor steps);
- Beginning on a separate line, a list of instructions in the IL language, the result of whose evaluation determines the transition condition;
- The terminating keyword END_TRANSITION on a separate line.

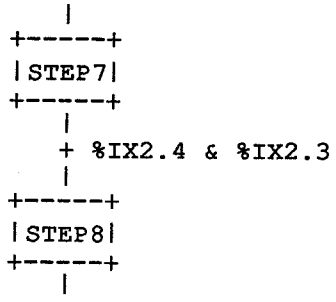
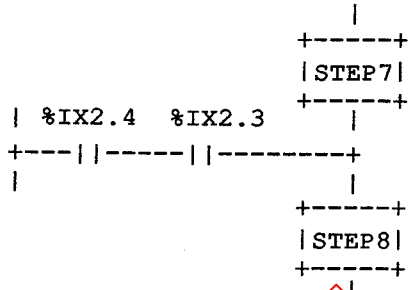
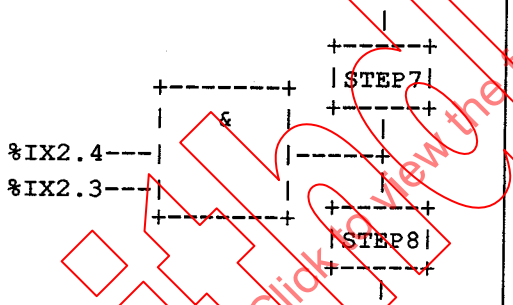
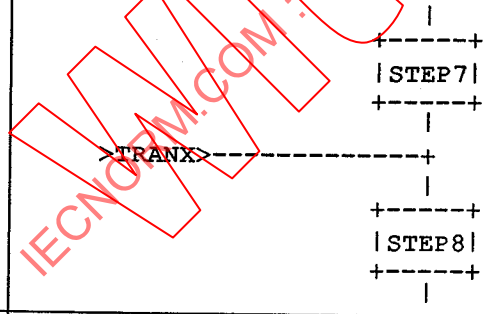
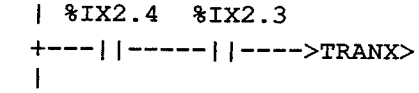
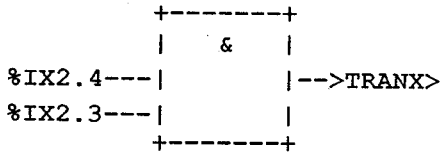
7) By the use of a *transition name* in the form of an identifier to the right of the directed link. This identifier shall refer to a TRANSITION...END_TRANSITION construction defining one of the following entities, whose evaluation shall result in the assignment of a Boolean value to the variable denoted by the transition name:

- A network in the LD or FBD language;
- A list of instructions in the IL language;
- An assignment of a Boolean expression in the ST language.

The *scope* of a transition name shall be *local* to the program organization unit in which the transition is located.

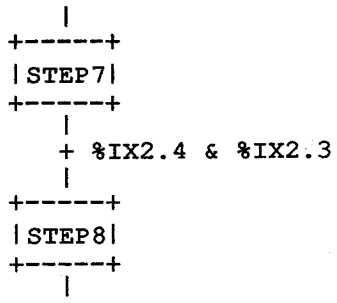
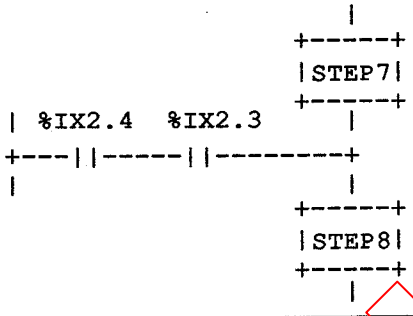
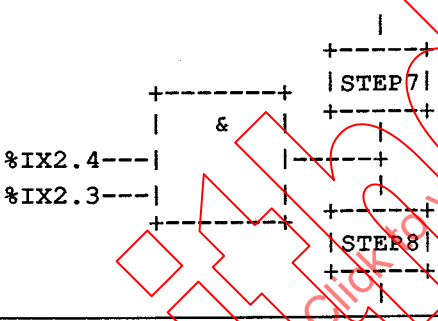
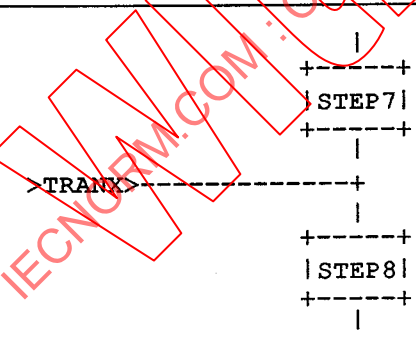
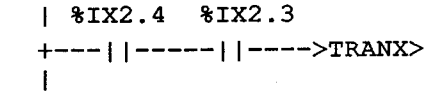
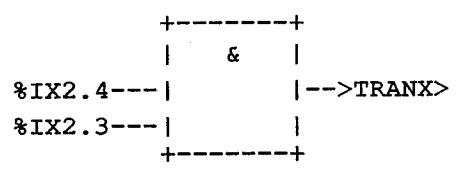
It shall be an *error* in the sense of 1.5.1 if any "side effect" (for instance, the assignment of a value to a variable other than the transition name) occurs during the evaluation of a transition condition.

Tableau 41 – Transitions et conditions de transition

N°	Exemple	Description
1		Etape précédente Condition de transition utilisant le langage ST (voir 3.3) Etape suivante
2		Etape précédente Condition de transition utilisant le langage LD (voir 4.2) Etape suivante
3		Etape précédente Condition de transition utilisant le langage FBD (voir 4.2) Etape suivante
4		Utilisation de connecteur: Etape précédente Connecteur de transition Etape suivante
4a		Condition de transition utilisant le langage LD (voir 4.2)
4b		Utilisant le langage FBD (voir 4.3)

(suite à la page 160)

Table 41 – Transitions and transition conditions

No.	Example	Description
1		Predecessor step Transition condition using ST language (see 3.3) Successor step
2		Predecessor step Transition condition using LD language (see 4.2) Successor step
3		Predecessor step Transition condition using FBD language (see 4.3) Successor step
4		Use of connector: Predecessor step Transition connector Successor step
4a		Transition condition: Using LD language (see 4.2)
4b		Using FBD language (see 4.3)

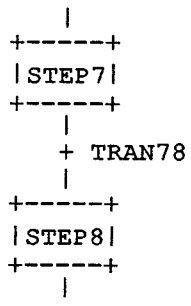
(continued on page 161)

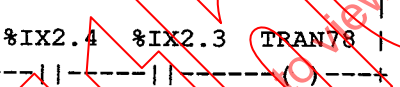
Tableau 41 *(fin)*

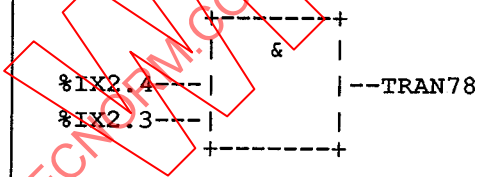
N°	Exemple	Description

5	`STEP STEP7 : END_STEP`
	`TRANSITION FROM STEP7 TO STEP 8`
	`  := %IX2.4 & %IX2.3 ;`
	`END_TRANSITION`
	`STEP STEP8 : END_STEP`
		Equivalent littéral de la caractéristique n° 1 utilisant le langage ST (voir 4.3)

6	`STEP STEP7 : END_STEP`
	`TRANSITION FROM STEP7 TO STEP 8:`
	`  LD %IX2.4`
	`  AND %IX2.3`
	`END_TRANSITION`
	`STEP STEP8 : END_STEP`
		Equivalent littéral de la caractéristique n° 1 utilisant le langage IL (voir 3.2)

7		Utilisation de nom de transition: Etape précédente Nom de transition Etape suivante

7a	`TRANSITION TRAN78 :`
	 
	`END_TRANSITION`
		Condition de transition utilisant le langage LD (voir 4.2)

7b	`TRANSITION TRAN78 :`
	 
	`END_TRANSITION`
		Condition de transition utilisant le langage FBD (voir 4.3)

7c	`TRANSITION TRAN78 :`
	`  LD %IX2.4`
	`  AND %IX2.3`
	`END_TRANSITION`
		Condition de transition utilisant le langage IL (voir 3.2)

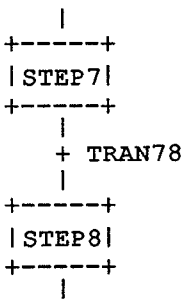
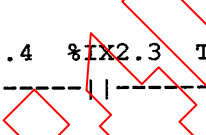
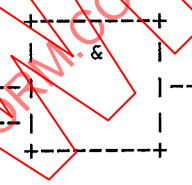
7d	`TRANSITION TRAN78 :`
	`  := %IX2.4 & %IX2.3 ;`
	`END_TRANSITION`
		Condition de transition utilisant le langage ST (voir 3.3)

NOTES

1 Si la caractéristique 1 du tableau 40 est acceptée, alors une ou plusieurs des caractéristiques 1, 2, 3, 4 ou 7 du présent tableau doivent être acceptées.

2 Si la caractéristique 2 du tableau 40 est acceptée, alors la caractéristique 5 ou 6 du présent tableau, ou les deux, doivent être acceptées.

Table 41 *(concluded)*

No.	Example	Description
5	STEP STEP7 : END_STEP TRANSITION FROM STEP7 TO STEP 8 := %IX2.4 & %IX2.3 ; END_TRANSITION STEP STEP8 : END_STEP	Textual equivalent of feature 1 using ST language (see 3.3)
6	STEP STEP7 : END_STEP TRANSITION FROM STEP7 TO STEP 8: LD %IX2.4 AND %IX2.3 END_TRANSITION STEP STEP8 : END_STEP	Textual equivalent of feature 1 using IL language (see 3.2)
7		Use of transition name: Predecessor step Transition name Successor step
7a	TRANSITION TRAN78 :  END_TRANSITION	Transition condition using LD language (see 4.2)
7b	TRANSITION TRAN78 :  END_TRANSITION	Transition condition using FBD language (see 4.3)
7c	TRANSITION TRAN78 : LD %IX2.4 AND %IX2.3 END_TRANSITION	Transition condition using IL language (see 3.2)
7d	TRANSITION TRAN78 : := %IX2.4 & %IX2.3 ; END_TRANSITION	Transition condition using ST language (see 3.3)

NOTES

1 If feature 1 of table 40 is supported, then one or more of features 1, 2, 3, 4, or 7 of this table shall be supported.

2 If feature 2 of table 40 is supported, then feature 5 or 6 of this table, or both, shall be supported.

2.6.4 *Actions*

Aucune action, une action ou plusieurs actions doivent être associées à chaque étape. Une étape qui n'a aucune action associée doit être considérée comme ayant une fonction WAIT, c'est-à-dire qu'elle attend qu'une condition de transition suivante devienne vraie.

Une action peut être une variable booléenne, un ensemble d'*instructions* dans le langage IL défini au paragraphe 3.2, un ensemble d'*énoncés* dans le langage ST défini en 3.3, un ensemble d'*échelons* dans le langage LD défini en 4.2, un ensemble de *réseaux* dans le langage FBD défini en 4.3, ou un diagramme fonctionnel en séquence (SFC) organisé conformément à 2.6.

Les actions doivent être déclarées par l'intermédiaire d'un ou de plusieurs des mécanismes définis en 2.6.4.1; elles doivent être associées à des étapes par l'intermédiaire de *corps d'étapes* littéraux ou de *blocs d'actions* graphiques, tels que définis en 2.6.4.2. Les détails relatifs à la représentation d'un bloc d'action sont définis en 2.6.4.3. La commande d'actions doit être exprimée par des *qualificatifs d'actions,* tels que définis en 2.6.4.4.

2.6.4.1 *Déclaration*

Une application d'automate programmable qui accepte des éléments de diagramme fonctionnel en séquence (SFC) doit fournir un ou plusieurs des mécanismes définis au tableau 42 pour les déclarations d'actions. Le *domaine* de la déclaration d'une action doit être *local* à l'unité d'organisation de programme contenant la déclaration.

2.6.4 *Actions*

Zero or more *actions* shall be associated with each step. A step which has zero associated actions shall be considered as having a WAIT function, that is, waiting for a successor transition condition to become true.

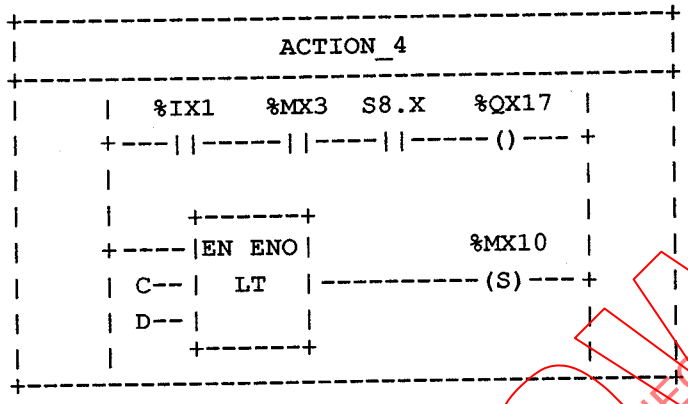
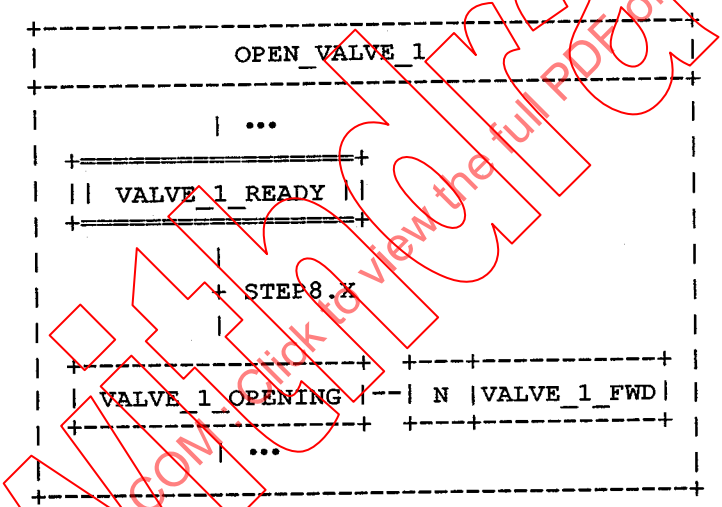
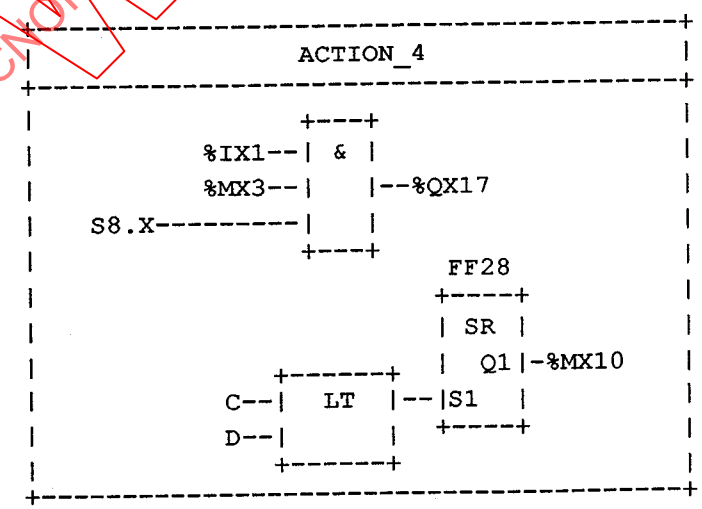
An action can be a Boolean variable, a collection of *instructions* in the IL language defined in 3.2, a collection of *statements* in the ST language defined in 3.3, a collection of *rungs* in the LD language defined in 4.2, a collection of *networks* in the FBD language defined in 4.3, or a *sequential function chart* (SFC) organized as defined in this subclause (2.6).

Actions shall be declared via one or more of the mechanisms defined in 2.6.4.1, and shall be associated with steps via textual *step bodies* or graphical *action blocks*, as defined in 2.6.4.2. The details of action block representation are defined in 2.6.4.3. Control of actions shall be expressed by *action qualifiers* as defined in 2.6.4.4.

2.6.4.1 *Declaration*

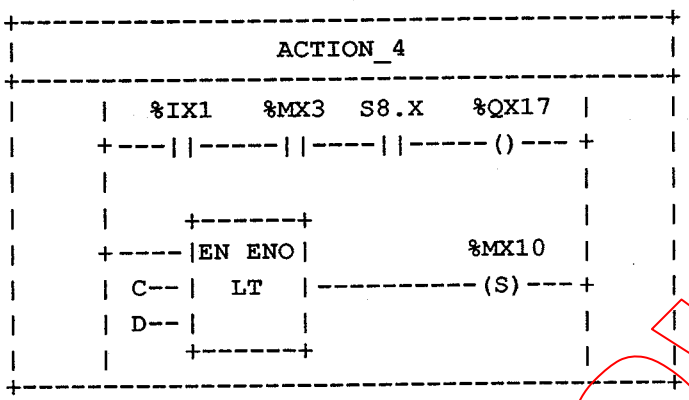
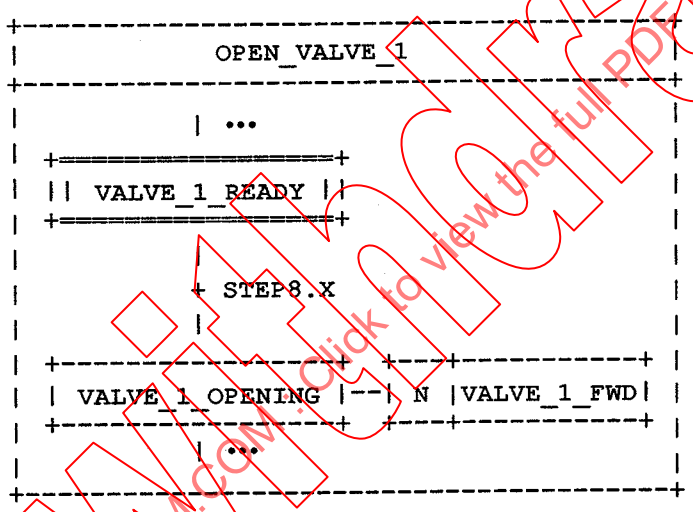
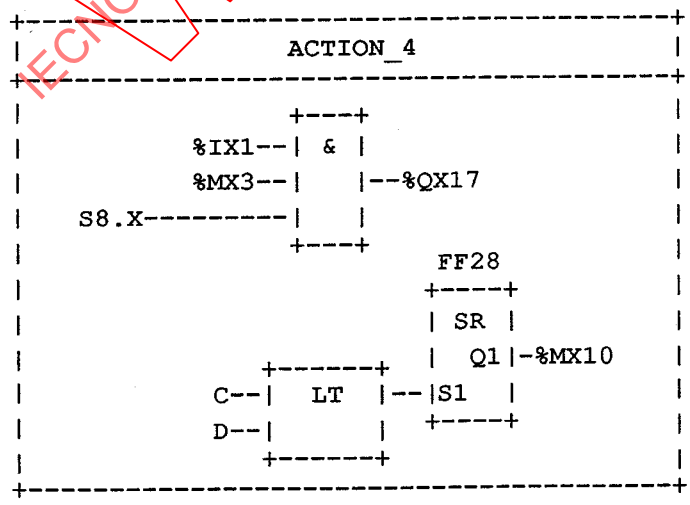
A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in table 42 for the declaration of actions. The *scope* of the declaration of an action shall be *local* to the program organization unit containing the declaration.

Tableau 42 – Déclaration d'actions

N°	Caractéristique	
1	Toute variable booléenne déclarée dans un bloc VAR ou VAR_OUTPUT, ou ses équivalents graphiques, peut être une action	
	Exemple	Caractéristique
2l		Déclaration graphique en langage LD (voir 4.2)
2s		Inclusion d'éléments SFC dans une action
2f		Déclaration graphique en langage FBD (voir 4.3)

(suite à la page 166)

Table 42 – Declaration of actions

No.	Feature	
1	Any Boolean variable in a VAR or VAR_OUTPUT block, or their graphical equivalents, can be an action	
	Example	Feature
2l		Graphical declaration in LD language (see 4.2)
2s		Inclusion of SFC elements in action
2f		Graphical declaration in FBD language (see 4.3)

(continued on page 167)

Tableau 42 *(fin)*

<table>
<tr><th>N°</th><th colspan="2">Caractéristique</th></tr>
<tr><td>1</td><td colspan="2">Toute variable booléenne déclarée dans un bloc VAR ou VAR_OUTPUT,
ou ses équivalents graphiques, peut être une action</td></tr>
<tr><td></td><td>Exemple</td><td>Caractéristique</td></tr>
<tr><td>3s</td><td><pre>ACTION ACTION_4 :
   %QX17 = %IX1 & %MX3 & S8.X ;
   FF28(S1 := (C<D)) ;
   %MX10 := FF28.Q ;
END_ACTION</pre></td><td>Déclaration littérale
en langage ST
(voir 3.3)</td></tr>
<tr><td>3i</td><td><pre>ACTION   ACTION_4 :
LD   S8.X
AND   %IX1
AND   %MX3
ST   %QX17
LD   C
LT   D
S1   FF28
LD   FF28.Q
ST   %MX10
END_ACTION</pre></td><td>Déclaration littérale
en langage IL
(voir 3.2)</td></tr>
<tr><td colspan="3">NOTES
1 Le drapeau d'étape S8.X est utilisé dans ces exemples pour obtenir le résultat souhaité selon lequel, lorsque S8 est désactivée, %QX17 := 0.
2 Si la caractéristique 1 du tableau 40 est acceptée, alors une ou plusieurs des caractéristiques de ce tableau, ou la caractéristique 4 du tableau 43 doivent être acceptées.
3 Si la caractéristique 2 du tableau 40 est acceptée, alors une ou plusieurs des caractéristiques 1, 3s, ou 3i de ce tableau doivent être acceptées.</td></tr>
</table>

Table 42 *(concluded)*

No.	Feature	
1	Any Boolean variable declared in a VAR or VAR_OUTPUT block, or their graphical equivalents, can be an action	
	Example	**Feature**

3s	`ACTION ACTION_4 :`
	`  %QX17 = %IX1 & %MX3 & S8.X ;`
	`  FF28(S1 := (C<D)) ;`
	`  %MX10 := FF28.Q ;`
	`END_ACTION`
		Textual declaration in ST language (see 3.3)

3i	`ACTION  ACTION_4 :`
	`LD  S8.X`
	`AND  %IX1`
	`AND  %MX3`
	`ST  %QX17`
	`LD  C`
	`LT  D`
	`S1  FF28`
	`LD  FF28.Q`
	`ST  %MX10`
	`END_ACTION`
		Textual declaration in IL language (see 3.2)

NOTES

1 The step flag S8.X is used in these examples to obtain the desired result that, when S8 is de-activated, %QX17 := 0.

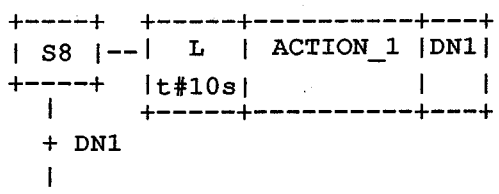
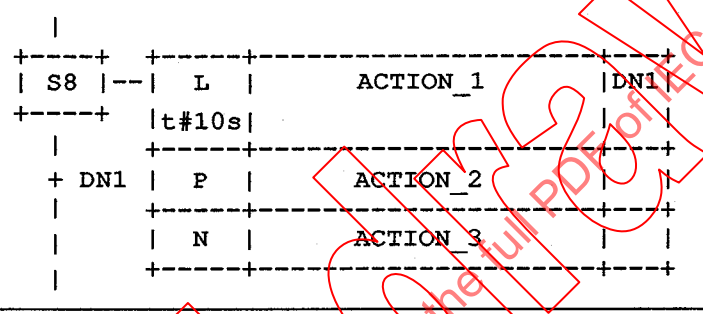
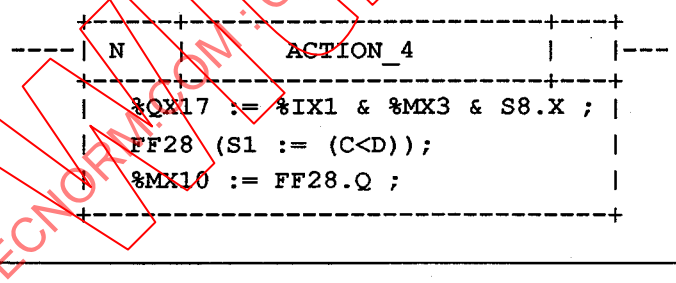
2 If feature 1 of table 40 is supported, then one or more of the features in this table, or feature 4 of table 43, shall be supported.

3 If feature 2 of table 40 is supported, then one or more of features 1, 3s, or 3i of this table shall be supported.

2.6.4.2 *Association avec des étapes*

Une application d'automate programmable, qui accepte des éléments de diagramme fonctionnel en séquence, doit fournir un ou plusieurs des mécanismes définis au tableau 43, pour l'association d'actions avec des étapes.

Tableau 43 – Association action/étape

N°	Exemple	Caractéristique
1		Bloc d'action (Voir 2.6.4.3)
2		Bloc d'action enchaînés
3	STEP S8 : ACTION_1(L,t#10s,DN1) ; ACTION_2(P) ; ACTION_3(N) ; END_STEP	Corps d'étape littéral
4		Bloc d'action champ "d" (Voir 2.6.4.3)

NOTE - Lorsque la caractéristique 4 est utilisée, le nom d'action correspondant ne peut être utilisé dans aucun autre bloc d'action.

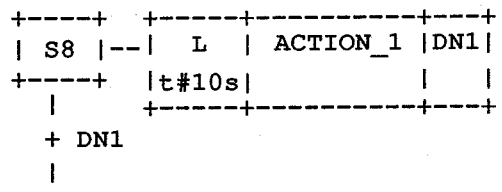
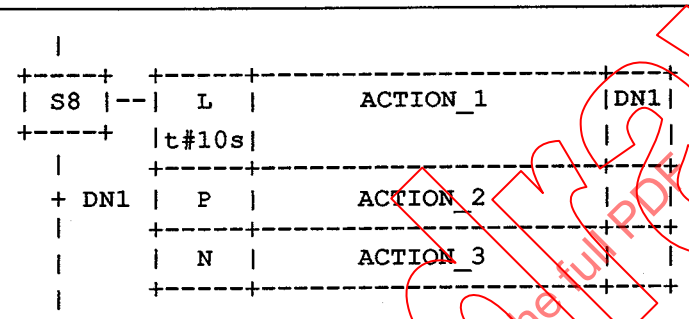
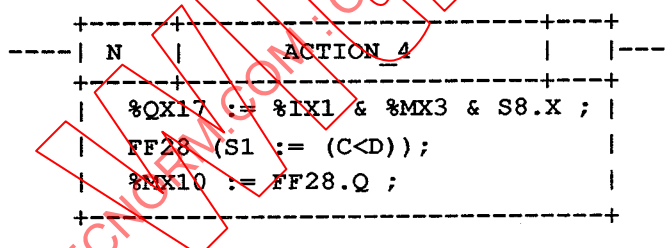
2.6.4.3 *Blocs d'actions*

Comme l'indique le tableau 44, un *bloc d'action* est un élément graphique pour la combinaison d'une variable booléenne avec un des *qualificatifs d'action* spécifié au 2.6.4.4, afin de générer une condition de validation, selon les règles données au 2.6.4.5, pour une action associée.

2.6.4.2 *Association with steps*

A programmable controller implementation which supports SFC elements shall provide one or more of the mechanisms defined in table 43 for the association of actions with steps.

Table 43 – Step/action association

No.	Example	Feature
1		Action block (see 2.6.4.3)
2		Concatenated action blocks
3	STEP S8 : ACTION_1(L,t#10s,DN1) ; ACTION_2(P) ; ACTION_3(N) ; END_STEP	Textual step body
4		Action block "d" Field (see 2.6.4.3)
NOTE - When feature 4 is used, the corresponding action name cannot be used in any other action block.		

2.6.4.3 *Action blocks*

As shown in table 44, an *action block* is a graphical element for the combination of a Boolean variable with one of the *action qualifiers* specified in subclause 2.6.4.4 to produce an enabling condition, according to the rules given in subclause 2.6.4.5, par an associated action.

Le bloc d'action fournit un moyen permettant de spécifier facultativement des variables "d'asservissement" booléennes, indiquées par la zone "c" dans le tableau 44; cette zone variable peut être sélectionnée par l'action spécifiée pour indiquer son achèvement, son délai d'attente, les conditions d'erreurs, etc. Si la zone "c" n'est pas présente et si la zone "b" spécifie que l'action doit être une variable booléenne, alors cette variable doit être interprétée comme la variable "c", lorsque cela est nécessaire.

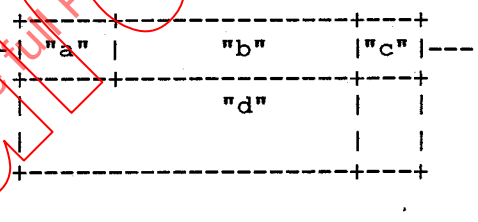
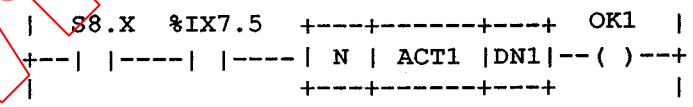
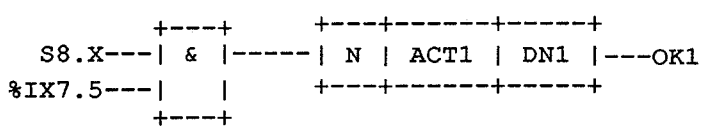
Lorsque des blocs d'actions sont enchaînés graphiquement comme l'indique le tableau 43, de tels enchaînements peuvent avoir plusieurs variables d'asservissement, mais ils ne doivent avoir qu'une seule variable d'entrée booléenne commune, qui doit agir simultanément sur tous les blocs enchaînés.

Comme il peut être également associé à une étape, un bloc d'action peut être utilisé comme un élément graphique dans le langage LD ou dans le langage FBD, définis à l'article 4. Dans ce cas, le flux du signal ou de la puissance, à travers un bloc d'action, doit suivre les règles spécifiées en 4.1.1.

Tableau 44 – Caractéristiques de bloc d'action

<table>
<tr><th>N°</th><th>Caractéristique</th><th>Forme graphique</th></tr>
<tr><td>1

2
3

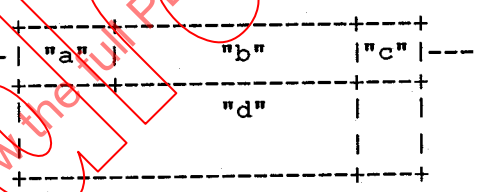
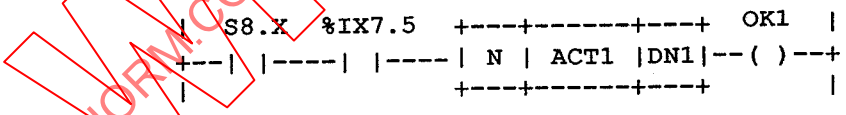
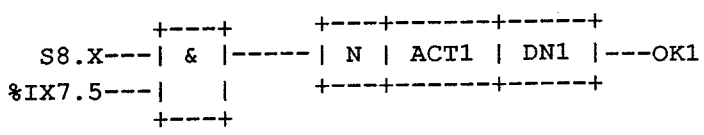
4
5
6
7</td><td>"a" : Qualificatif conforme à 2.6.4.4
"b" : Nom d'action
"c" : Variables booléennes d'asservissement
"d" : Action utilisant:
le langage IL (3.2)
le langage ST (3.3)
le langage LD (4.2)
le langage FBD (4.3)</td><td>  </td></tr>
<tr><td></td><td colspan="2">Caractéristique/Exemple</td></tr>
<tr><td>8</td><td colspan="2">Utilisation de blocs d'actions en schémas à contacts (voir 4.2):
  </td></tr>
<tr><td>9</td><td colspan="2">Utilisation de blocs d'actions en schémas de blocs fonctionnels (voir 4.3):
  </td></tr>
<tr><td colspan="3">NOTES
1 Le champ "a" peut être omis lorsque le qualificatif est "N".
2 Le champ "c" peut être omis lorsqu'aucune variable d'asservissement n'est utilisée.</td></tr>
</table>

The action block provides a means of optionally specifying Boolean "indicator" variables, indicated by the "c" field in table 44, which can be set by the specified action to indicate its completion, timeout, error conditions, etc. If the "c" field is not present, and the "b" field specifies that the action shall be a Boolean variable, then this variable shall be interpreted as the "c" variable when required.

When action blocks are concatenated graphically as illustrated in table 43, such concatenations can have multiple indicator variables, but shall have only a single common Boolean input variable, which shall act simultaneously upon all the concatenated blocks.

As well as being associated with a step, an action block can be used as a graphical element in the LD or FBD languages specified in clause 4. In this case, signal or power flow through an action block shall follow the rules specified in 4.1.1.

Table 44 – Action block features

No.	Feature	Graphical form
1	"a" : Qualifier as per 2.6.4.4	
2	"b" : Action name	
3	"c" : Boolean "indicator" variables	
	"d" : Action using:	
4	IL language (3.2)	
5	ST language (3.3)	
6	LD language (4.2)	
7	FBD language (4.3)	
	Feature/Example	
8	Use of action blocks in ladder diagrams (see 4.2):	
9	Use of action blocks in function block diagrams (see 4.3):	

NOTES

1 Field "a" can be omitted when the qualifier is "N".

2 Field "c" can be omitted when no indicator variable is used.

2.6.4.4 *Qualificatifs d'actions*

Un *qualificatif d'action* doit être associé à chaque association étape/action définie en 2.6.4.2, ou à chaque apparition d'un bloc d'action tel que défini en 2.6.4.3. La valeur de ce qualificatif doit être égale à l'une des valeurs énumérées au tableau 45. En outre, les qualificatifs L, D, SD, DS, et SL doivent avoir une durée associée de type TIME.

NOTE - La CEI 848 fournit des définitions et des exemples d'utilisation de ces qualificatifs. La présente norme donne un caractère formel à ces définitions, en redéfinissant le qualificatif "S" et en introduisant le qualificatif "R". La commande d'actions utilisant ces qualificatifs est définie au paragraphe suivant, et des exemples supplémentaires de leur utilisation sont donnés à l'annexe F.

Tableau 45 – Qualificatifs d'action

N°	Qualificatif	Explication
1	Néant	Non mémorisé (qualificatif nul)
2	N	Non mémorisé
3	R	Remise à zéro prioritaire
4	S	Positionné (mémorisé)
5	L	Limité dans le temps
6	D	Temporisé
7	P	Impulsion
8	SD	Mémorisé et temporisé
9	DS	Temporisé et mémorisé
10	SL	Mémorisé et limité dans le temps

2.6.4.5 *Commande d'actions*

La commande d'actions doit être équivalente, au point de vue fonctionnel, à l'application des règles suivantes:

1) A chaque action doit être associé l'équivalent fonctionnel d'une instance du bloc fonctionnel ACTION_CONTROL défini aux figures 14 et 15. Si l'action est déclarée comme étant une variable booléenne, telle que définie en 2.6.4.1, la sortie "Q" de ce bloc doit être l'état de cette variable booléenne. Si l'action est déclarée comme étant un ensemble d'énoncés ou de réseaux, tels que définis en 2.6.4.1, alors cet ensemble doit être exécuté en continu, tant que la sortie "Q" du bloc fonctionnel ACTION_CONTROL reste à la valeur booléenne 1. Les énoncés ou les réseaux doivent être exécutés une fois pour toutes après le front descendant de "Q".

2) Il est nécessaire de considérer qu'une entrée booléenne au bloc ACTION_CONTROL, relatif à une action, a une *association* avec une étape telle que définie en 2.6.4.2, ou avec un bloc d'action tel que défini en 2.6.4.3, si le qualificatif correspondant est équivalent au nom d'entrée (N, R, S, L, D, P, SD, DS, ou SL). L'association doit être considérée comme *active* si l'étape associée est active, ou si l'entrée du bloc d'action associé a la valeur booléenne 1. *Les associations actives* d'une *action* sont équivalentes à l'ensemble des *associations actives* de toutes les entrées à son bloc fonctionnel ACTION_CONTROL.

2.6.4.4 *Action qualifiers*

Associated with each step/action association defined in 2.6.4.2, or each occurrence of an action block as defined in 2.6.4.3, shall be an *action qualifier*. The value of this qualifier shall be one of the values listed in table 45. In addition, the qualifiers L, D, SD, DS, and SL shall have an associated duration of type TIME.

NOTE - IEC 848 gives informal definitions and examples of the use of these qualifiers. This standard formalizes these definitions, redefining the "S" qualifier and introducing the "R" qualifier. The control of actions using these qualifiers is defined in the following subclause, and additional examples of their use are given in annex F.

Table 45 – Action qualifiers

No.	Qualifier	Explanation
1	None	Non-stored (null qualifier)
2	N	Non-stored
3	R	overriding Reset
4	S	Set (Stored)
5	L	time Limited
6	D	time Delayed
7	P	Pulse
8	SD	Stored and time Delayed
9	DS	Delayed and Stored
10	SL	Stored and time Limited

2.6.4.5 *Action control*

The control of actions shall be functionally equivalent to the application of the following rules:

1) Associated with each action shall be the functional equivalent of an instance of the ACTION_CONTROL function block defined in figures 14 and 15. If the action is declared as a Boolean variable, as defined in 2.6.4.1, the "Q" output of this block shall be the state of this Boolean variable. If the action is declared as a collection of statements or networks, as defined in 2.6.4.1, then this collection shall be executed continually while the "Q" output of the ACTION_CONTROL function block stands at Boolean 1. The statements or networks shall be executed one final time after the falling edge of "Q".

2) A Boolean input to the ACTION_CONTROL block for an action shall be said to have an *association* with a step as defined in 2.6.4.2, or with an action block as defined in 2.6.4.3, if the corresponding qualifier is equivalent to the input name (N, R, S, L, D, P, SD, DS, or SL). The association shall be said to be *active* if the associated step is active, or if the associated action block's input has the value Boolean 1. The *active associations* of an *action* are equivalent to the set of *active associations* of all inputs to its ACTION_CONTROL function block.

Une entrée booléenne vers un bloc ACTION_CONTROL doit avoir la valeur booléenne 1 si elle a au moins une *association active,* et la valeur booléenne 0 dans le cas contraire.

3) La valeur de l'entrée T vers un bloc ACTION_CONTROL doit être égale à la valeur de la portion de durée d'un qualificatif temporel (L, D, SD, DS, ou SL) d'une *association active.* Si aucune association de ce type n'existe, la valeur de l'entrée T doit être t#0s.

4) L'existence d'une ou de plusieurs des conditions suivantes doit être une *erreur,* au sens de 1.5.1:

a) plus d'une *association active* d'une action a un qualificatif temporel (L, D, SD, DS, ou SL);

b) L'entrée SD d'un bloc ACTION_CONTROL a la valeur booléenne 1 lorsque la sortie Q1 de son bloc SL_FF a la valeur booléenne 1;

c) La sortie SL vers un bloc ACTION_CONTROL a la valeur booléenne 1 lorsque la sortie Q1 de son bloc SD_FF a la valeur booléenne 1.

5) Il n'est pas nécessaire que le bloc ACTION_CONTROL lui-même soit mis en oeuvre; il est uniquement nécessaire que la commande d'actions soit équivalente aux règles précédentes. Comme l'illustre la figure 16., seules les parties de la commande d'action relatives à une action particulière doivent être instanciées. En particulier, il convient de noter que la fonction simple MOVE (:=) et la fonction booléenne OR suffisent pour la commande d'actions à variables booléennes, si l'association de ces dernières a uniquement des qualificatifs "N".

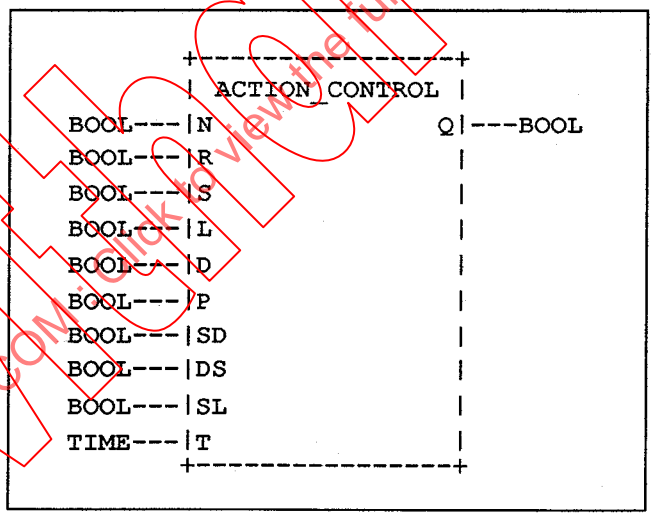


Figure 14 – Bloc fonctionnel ACTION_CONTROL – Interface externe (non visible par l'utilisateur)

A Boolean input to an ACTION_CONTROL block shall have the value Boolean 1 if it has at least one active association, and the value Boolean 0 otherwise.

3) The value of the T input to an ACTION_CONTROL block shall be the value of the duration portion of a time-related qualifier (L, D, SD, DS, or SL) of an active association. If no such association exists, the value of the T input shall be t#0s.

4) It shall be an *error* in the sense of subclause 1.5.1 if one or more of the following conditions exist:

a) More than one *active association* of an action has a time-related qualifier (L, D, SD, DS, or SL).

b) The SD input to an ACTION_CONTROL block has the Boolean value 1 when the Q1 output of its SL_FF block has the Boolean value 1.

c) The SL input to an ACTION_CONTROL block has the Boolean value 1 when the Q1 output of its SD_FF block has the Boolean value 1.

5) It is not required that the ACTION_CONTROL block itself be implemented, but only that the control of actions be equivalent to the preceding rules. Only those portions of the action control appropriate to a particular action need be instantiated, as illustrated in figure 16. In particular, note that simple MOVE (:=) and Boolean OR functions suffice for control of Boolean variable actions if the latter's associations have only "N" qualifiers.

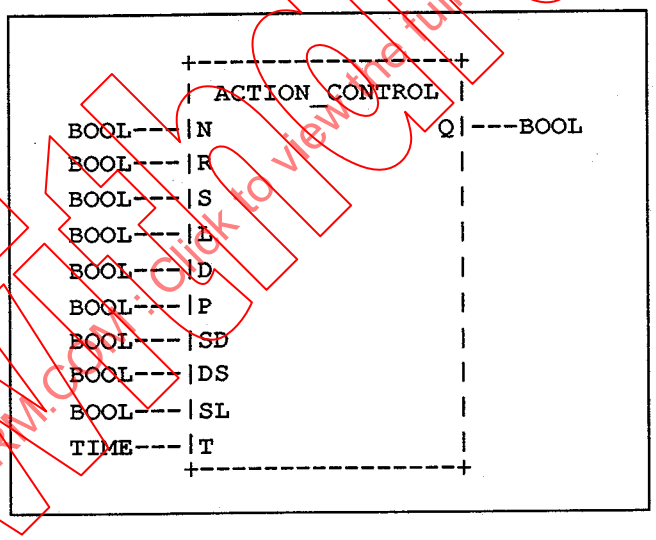


Figure 14 – ACTION_CONTROL function block – External interface (not visible to the user)

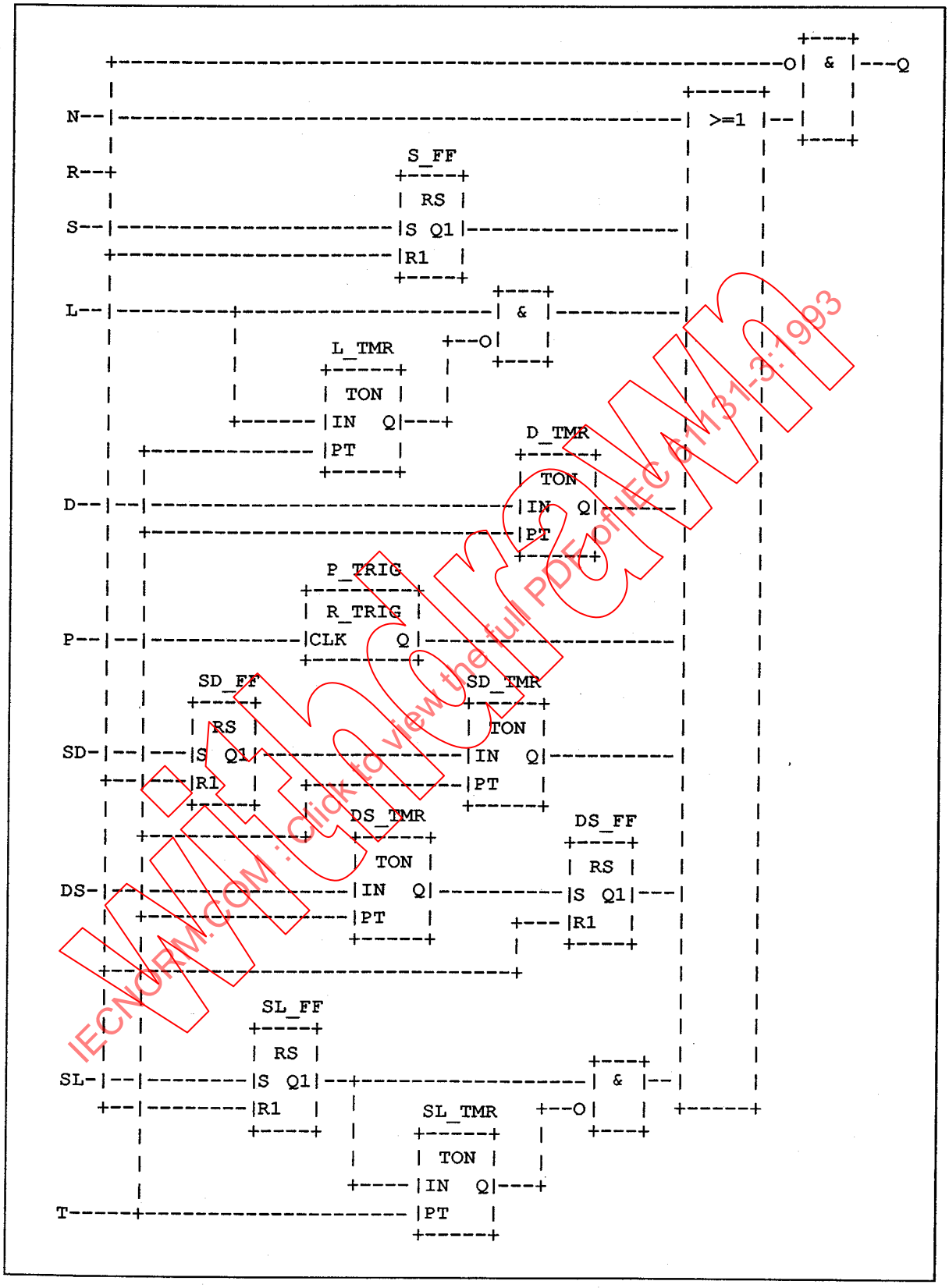


Figure 15 – Corps de bloc fonctionnel ACTION_CONTROL
(non visible par l'utilisateur)

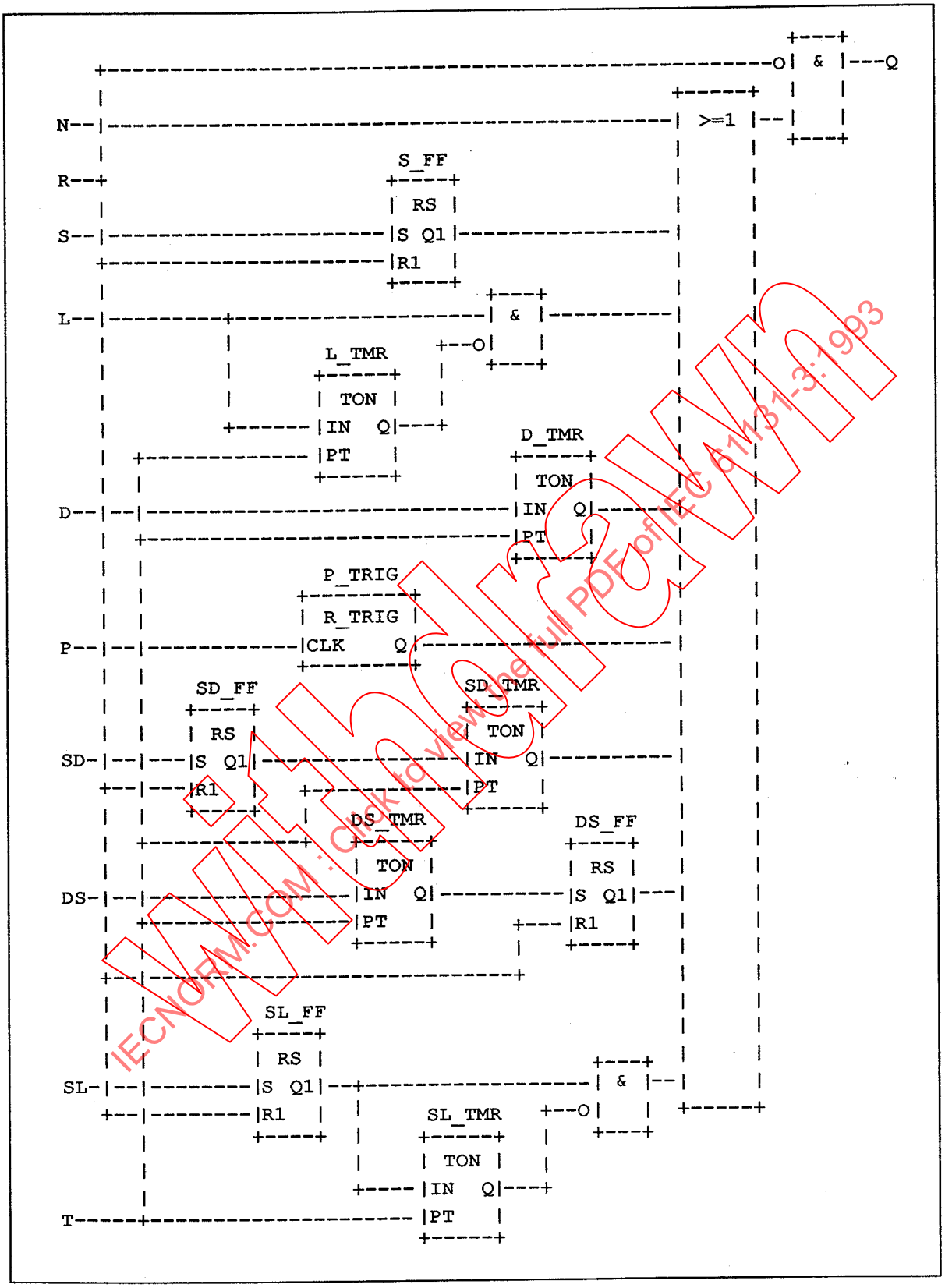
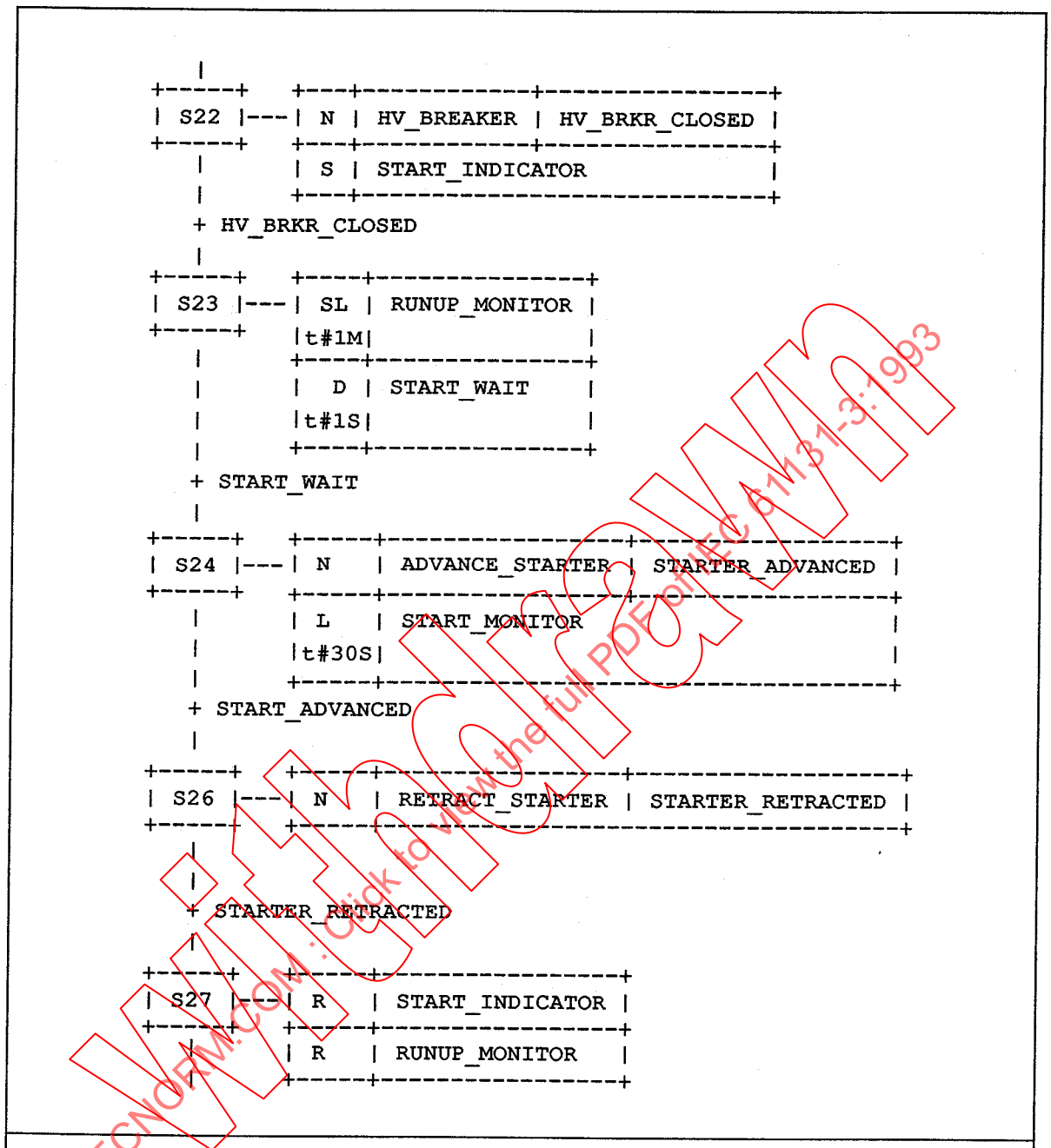
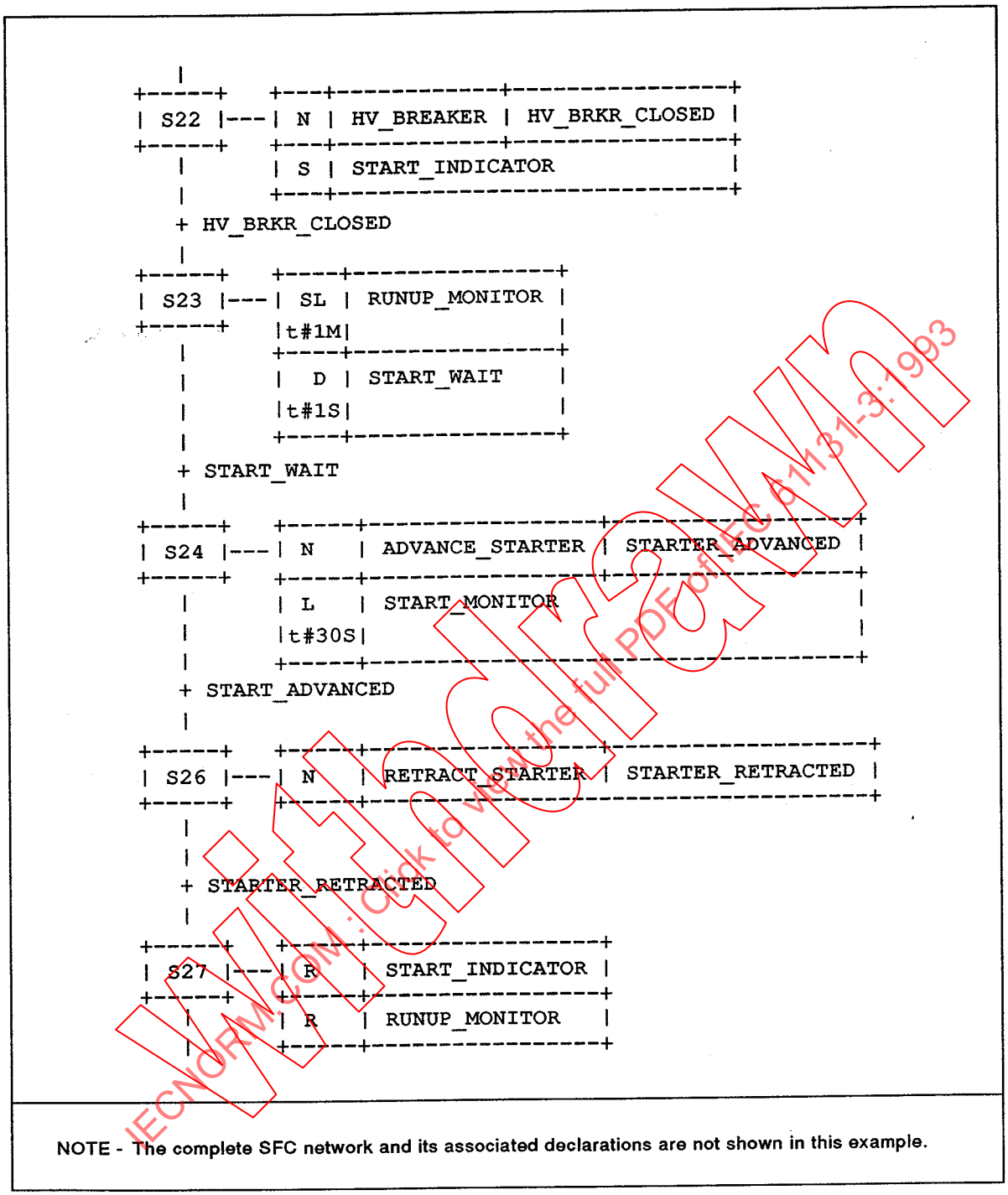


Figure 15 – ACTION_CONTROL function block body
(not visible to the user)



NOTE – Le réseau de diagramme fonctionnel en séquence (SFC) complet et ses déclarations associées ne sont pas indiqués dans cet exemple.

Figure 16a – Exemple de commande d'action – Représentation par diagramme fonctionnel en séquence (SFC)



NOTE - The complete SFC network and its associated declarations are not shown in this example.

Figure 16a – Action control example – SFC representation

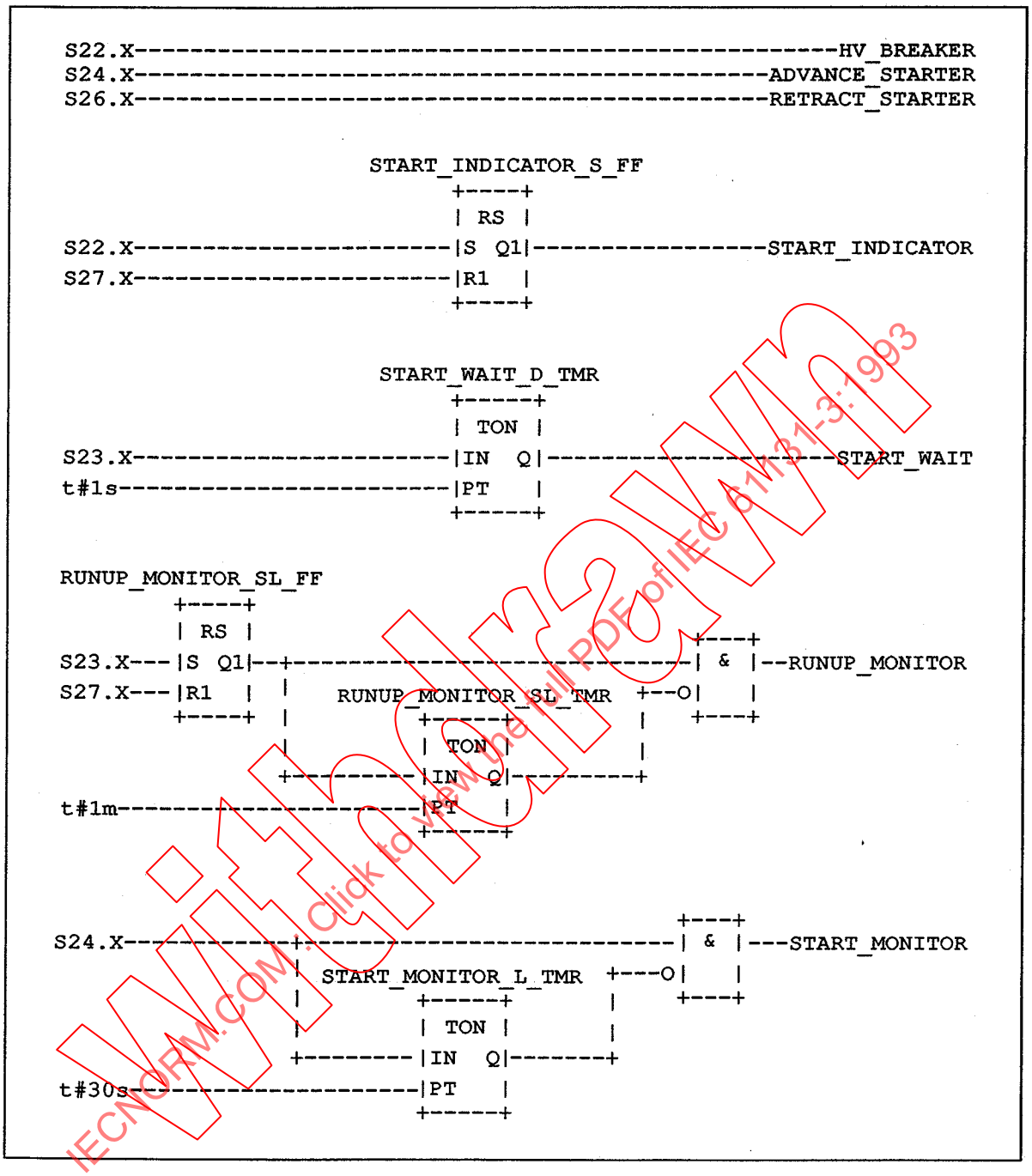


Figure 16b – Exemple de commande d'action – Equivalent fonctionnel

2.6.5 *Règles d'évolution*

La *situation initiale* d'un réseau de diagramme fonctionnel en séquence (SFC) est caractérisée par l'*étape initiale* qui se trouve à l'état actif lors de l'initialisation du programme ou du bloc fonctionnel contenant le réseau.

Des *évolutions* des états actifs doivent avoir lieu le long des *liaisons dirigées* lorsqu'elles sont provoquées par l'*effacement* d'une ou de plusieurs *transitions*.

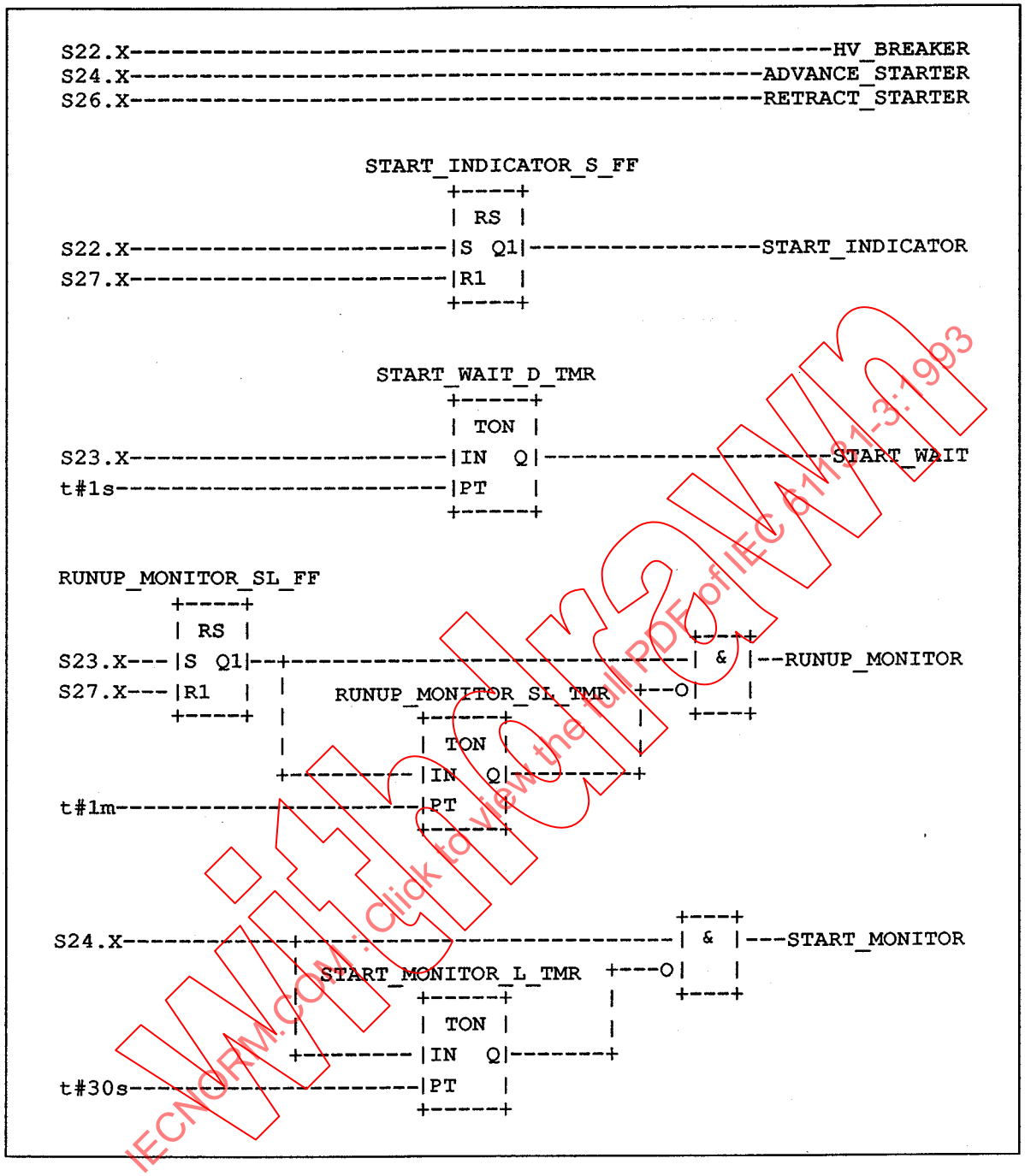


Figure 16b – Action control example – Functional equivalent

2.6.5 *Rules of evolution*

The *initial situation* of a SFC network is characterized by the *initial step* which is in the active state upon initialization of the program or function block containing the network.

Evolutions of the active states of steps shall take place along the *directed links* when caused by the *clearing* of one or more *transitions*.

Une transition est *validée* lorsque toutes les étapes précédentes, reliées au symbole de transition correspondant par des liaisons dirigées, sont actives. L'effacement d'une transition se produit lorsque la transition est validée et lorsque la condition de transition associée est vraie.

L'effacement d'une transition entraîne la *désactivation* (ou "la remise à zéro") de toutes étapes immédiatement précédentes reliées au symbole de transition correspondant par des liaisons dirigées, suivie de l'*activation* de toutes les étapes qui suivent immédiatement.

L'alternance étape/transition et transition/étape doit toujours être maintenue dans des connexions d'éléments de diagramme fonctionnel en séquence (SFC), c'est-à-dire:

– deux étapes ne doivent jamais être reliées directement; elles doivent toujours être séparées par une transition;

– Deux transitions ne doivent jamais être directement reliées; elles doivent toujours être séparées par une étape.

Lorsque l'effacement d'une transition conduit à l'activation de plusieurs étapes en même temps, les séquences auxquelles ces étapes appartiennent sont appelées *séquences simultanées.* Après leur activation simultanée, l'évolution de chacune de ces séquences devient indépendante. Afin de mettre l'accent sur la nature spéciale de telles constructions, la divergence et la convergence de séquences simultanées doivent être indiquées par une double ligne horizontale.

Le tableau 46 définit la syntaxe et la sémantique des combinaisons d'étapes et de transitions.

Théoriquement, il est possible de choisir la longueur du temps d'effacement d'une transition, mais ce temps ne doit jamais être égal à zéro. Dans la pratique, le temps d'effacement sera imposé par l'application de l'automate programmable. Pour la même raison, la durée de l'activité d'une étape ne peut jamais être considérée comme égale à zéro.

Plusieurs transitions, qui peuvent être effacées simultanément, doivent être effacées simultanément, dans les limites de temps de l'application particulière d'automate programmable ainsi que dans les limites des contraintes de priorité définies au tableau 46.

Les essais relatifs à la ou aux conditions de transition suivantes d'une étape active, ne doivent pas être effectués, tant que les effets de l'activation de l'étape ne se sont pas propagés à travers toute l'unité d'organisation de programme dans laquelle l'étape est déclarée.

La figure 17 illustre l'application des règles précédemment évoquées. Dans cette figure, l'état actif d'une étape est indiqué par la présence d'un astérisque (*) dans le bloc correspondant. Cette notation est uniquement utilisée pour l'illustration, et ne constitue pas une caractéristique de langage requise.

L'application des règles données dans le présent paragraphe ne peut empêcher la formation d'éléments de diagramme fonctionnel en séquence "incertains", tel que celui illustré à la figure 18a, qui peut donner lieu à une prolifération incontrôlée de jetons. De la même manière, l'application de ces règles ne peut empêcher l'application d'éléments de diagramme fonctionnel en séquence "inaccessibles", tel que celui illustré à la figure 18b, qui peut donner lieu à un comportement "bloqué". Le système d'automate programmable doit traiter l'existence de telles conditions comme des *erreurs,* telles que définies en 1.5.1.

A transition is *enabled* when all the preceding steps, connected to the corresponding transition symbol by directed links, are active. The clearing of a transition occurs when the transition is enabled and when the associated transition condition is true.

The clearing of a transition causes the *deactivation* (or "resetting") of all the immediately preceding steps connected to the corresponding transition symbol by directed links, followed by the *activation* of all the immediately following steps.

The alternation Step/Transition and Transition/Step shall always be maintained in SFC element connections, that is:

– Two steps shall never be directly linked; they shall always be separated by a transition.

– Two transitions shall never be directly linked; they shall always be separated by a step.

When the clearing of a transition leads to the activation of several steps at the same time, the sequences to which these steps belong are called *simultaneous sequences*. After their simultaneous activation, the evolution of each of these sequences becomes independent. In order to emphasize the special nature of such constructs, the divergence and convergence of simultaneous sequences shall be indicated by a double horizontal line.

Table 46 defines the syntax and semantics of the allowed combinations of steps and transitions.

The clearing time of a transition may theoretically be considered as short as one may wish, but it can never be zero. In practice, the clearing time will be imposed by the programmable controller implementation. For the same reason, the duration of a step activity can never be considered to be zero.

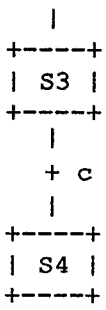
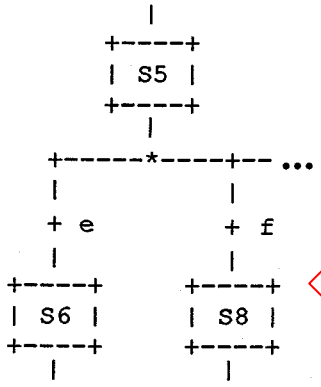
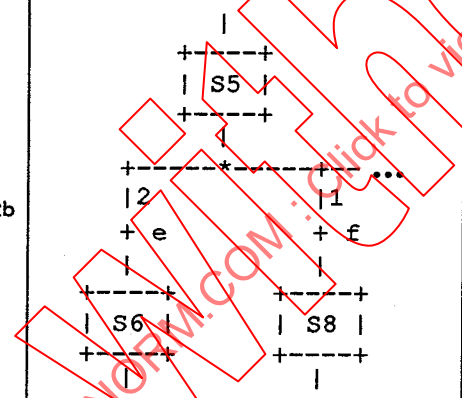
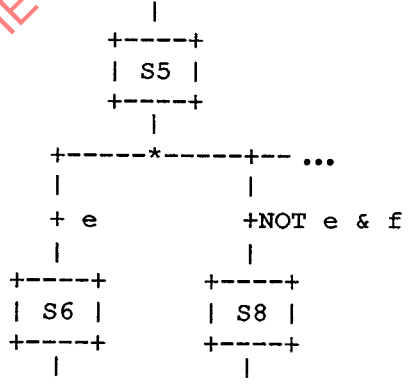
Several transitions which can be cleared simultaneously shall be cleared simultaneously, within the timing constraints of the particular programmable controller implementation and the priority constraints defined in table 46.

Testing of the successor transition condition(s) of an active step shall not be performed until the effects of the step activation have propagated throughout the program organization unit in which the step is declared.

Figure 17 illustrates the application of these rules. In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

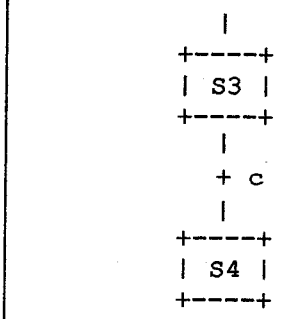
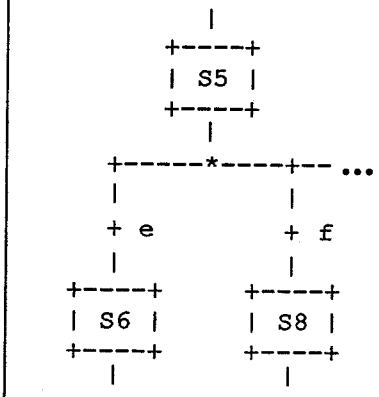
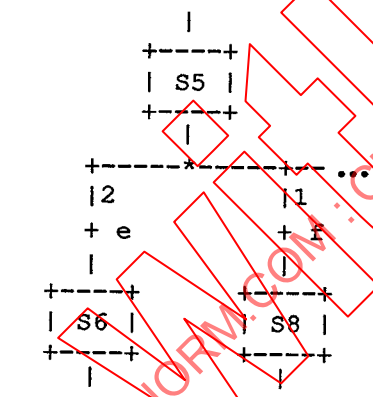
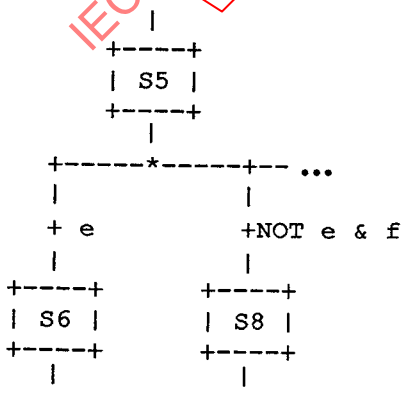
The application of the rules given in this subclause cannot prevent the formulation of "unsafe" SFCs, such as the one shown in figure 18a, which may exhibit uncontrolled proliferation of tokens. Likewise, the application of these rules cannot prevent the formulation of "unreachable" SFCs, such as the one shown in figure 18b, which may exhibit "locked up" behavior. The programmable controller system shall treat the existence of such conditions as *errors* as defined in 1.5.1.

Tableau 46 – Evolution de séquence

N°	Exemple	Règle
1		*Séquence unique:* L'alternance Etape-Transition est répétée en série. *Exemple:* Une évolution de l'étape S3 à l'étape S4 doit se produire si et uniquement si l'étape S3 se trouve dans l'état actif et que la condition de transition c est vraie.
2a		*Divergence de sélection de séquence:* Une sélection entre plusieurs séquences est représentée par autant de symboles de transition, *sous* la ligne horizontale, qu'il y a d'évolutions différentes possibles. L'astérisque indique des évaluations de priorité de transition de gauche à droite. *Exemple:* Une évolution de S5 à S6 doit uniquement se produire si S5 est active et si la condition de transition "e" est vraie, une évolution de S5 à S8 doit uniquement se produire si S5 est active et si "f" est vraie et "e" est fausse.
2b		*Divergence de sélection de séquence:* L'astérisque, suivi de branches numérotées, indique une priorité, définie par l'utilisateur, d'évaluation de transition; la branche portant le plus faible numéro ayant le niveau de priorité le plus élevé. *Exemple:* Une évolution de S5 à S8 doit uniquement se produire si S5 est active et si la condition de transition "f" est vraie, une évolution de S5 à S6 doit uniquement se produire si S5 est active, "e" est vraie et "f" est fausse.
2c		*Divergence de sélection de séquence:* La connexion de la branche indique que l'utilisateur doit assurer que les conditions de transition soient mutuellement exclusives, comme spécifié dans la CEI 848. *Exemple:* S6 que si S5 est active et la condition de transitions "e" est vraie, ou de S5 à S8 que si S5 est active et "e" est fausse et "f" est vraie.

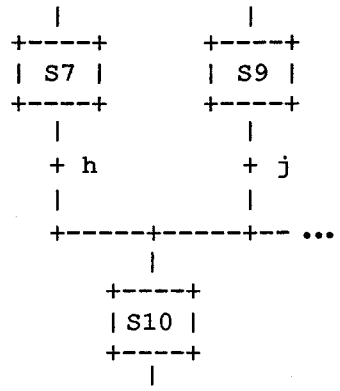
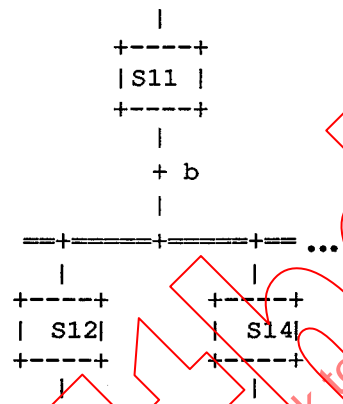
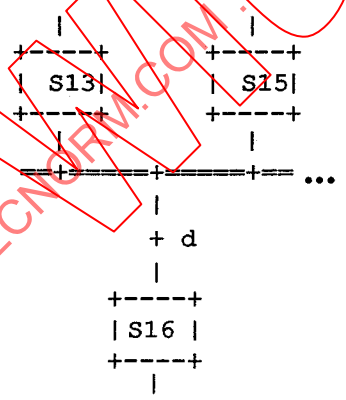
(suite à la page 186)

Table 46 – Sequence evolution

No.	Example	Rule
1		*Single sequence:* The alternation step-transition is repeated in series. *Example:* An evolution from step S3 to step S4 shall take place if and only if step S3 is in the active state and the transition condition c is true.
2a		*Divergence of sequence selection:* A selection between several sequences is represented by as many transition symbols, *under* the horizontal line, as there are different possible evolutions. The asterisk denotes left-to-right priority of transition evaluations. *Example:* An evolution shall take place from S5 to S6 only if S5 is active and the transition condition "e" is true, or from S5 to S8 only if S5 is active and "f" is true and "e" is false.
2b		*Divergence of sequence selection:* The asterisk, followed by numbered branches, indicates a user-defined priority of transition evaluation, with the lowerst-numbered branch having the highest priority. *Example:* An evolution shall take place from S5 to S8 only if S5 is active and the transition condition "f" is true, or from S5 to S6 only if S5 is active, and "e" is true, and "f" is false.
2c		*Divergence of sequence selection:* The connection of the branch indicates that the user must assure that transition conditions are mutually exclusive, as specified by IEC 848. *Exemple:* S6 only if S5 is active and the transition condition "e" is true, or from S5 to S8 only if S5 is active and "e" is false and "f" is true.

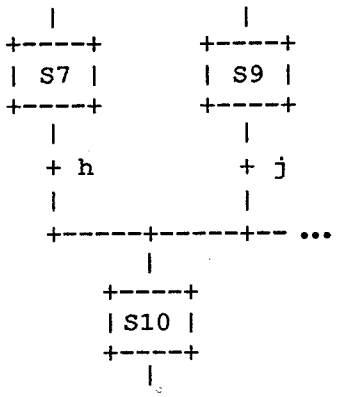
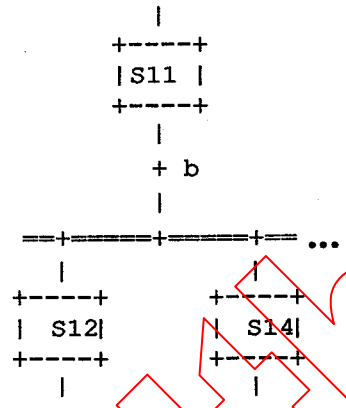
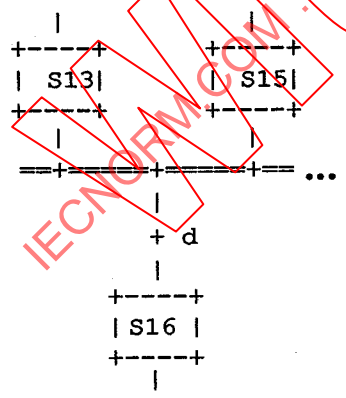
(continued on page 187)

Tableau 46 *(suite)*

N°	Exemple	Règle
3		*Convergence de sélection de séquence:* La fin d'une sélection de séquence est représentée par autant de symboles de transition, *au-dessus* de la ligne horizontale, qu'il y a de chemins de sélection à terminer. *Exemple:* Une évolution de S7 à S10 doit uniquement se produire si S7 est active et si la condition de transition "h" est vraie; une évolution de S9 à S10 doit uniquement se produire si S9 est active et si "j" est vraie.
4		*Séquences simultanées – Divergence:* Un seul symbole de transition commun doit être possible, immédiatement *au-dessus* de la double ligne horizontale de synchronisation. *Exemple:* Une évolution de S11 à S12, S14, ... doit uniquement se produire si S11 est active et si la condition de transition "b", associée à la transition commune, est vraie. Après l'activation simultanée de S12, S14, etc., l'évolution de chaque séquence se dévoile indépendamment.
		*Séquences simultanées – Convergence:* Un seul symbole commun de transition doit être possible, immédiatement *au-dessous* de la double ligne horizontale de synchronisation. *Exemple:* Une évolution de S13, S15, ... à S16 doit uniquement se produire si toutes les étapes situées au-dessus et reliées à la double ligne horizontale, sont actives et si la condition de transition "d", associée à la transition commune, est vraie.

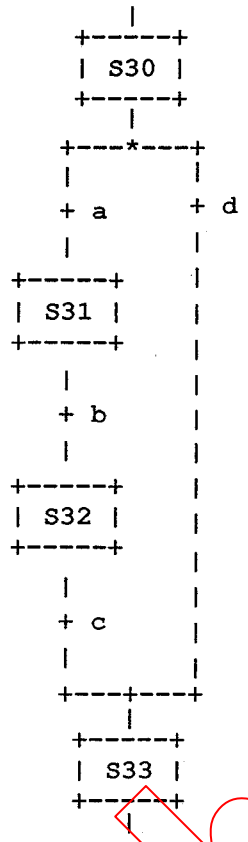
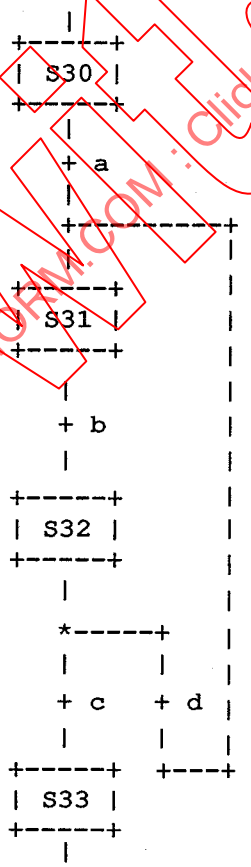
(suite à la page 188)

Table 46 *(continued)*

No.	Example	Rule
3		*Convergence of sequence selection:* The end of a sequence selection is represented by as many transition symbols, *above* the horizontal line, as there are selection paths to be ended. *Example:* An evolution shall take place from S7 to S10 only if S7 is active and the transition "h" is true, or from S9 to S10 only if S9 is active and "j" is true.
4		*Simultaneous sequences – divergence:* Only one common transition symbol shall be possible, immediately *above* the double horizontal line of synchronization. *Example:* An evolution shall take place from S11 to S12, S14, ... only if S11 is active and the transition condition "b" associated to the common transition is true. After the simultaneous activation of S12, S14, etc., the evolution of each sequence proceeds independently.
		*Simultaneous sequences – convergence:* Only one common transition symbol shall be possible, immediately *under* the double horizontal line of synchronization. *Example:* An evolution shall take place from S13, S15, ... to S16 only if all steps above and connected to the double horizontal line are active and the transition condition "d" associated to the common transition is true.

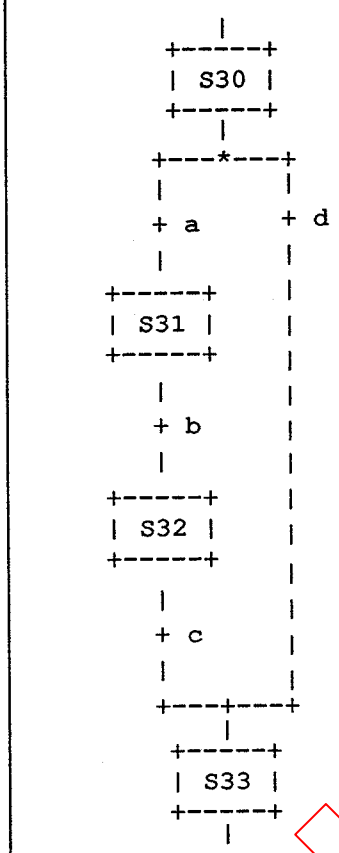
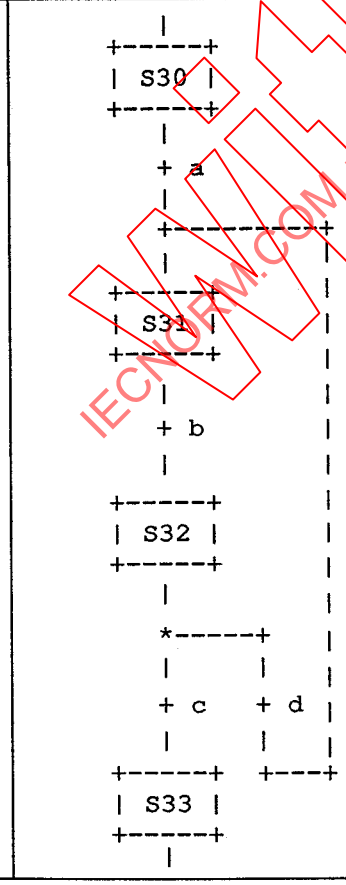
(continued on page 189)

Tableau 46 *(suite)*

N°	Exemple	Règle
5a 5b 5c		*Saut de séquence:* Un "saut de séquence" est un cas spécial de sélection de séquence (caractéristique 2) dans lequel une ou plusieurs branches ne contiennent aucune étape. Les caractéristiques 5a, 5b et 5c correspondent aux options de représentation données respectivement dans les caractéristiques 2a, 2b et 2c. *Exemple:* (Caractéristique 5a illustrée) Une évolution, de S30 à S33, doit uniquement se produire si "a" est fausse et si "d" est vraie, c'est-à-dire que la séquence (S31, S32) sera sautée.
6a 6b 6c		*Boucle de séquence:* Une "boucle de séquence" est un cas spécial de sélection de séquence (caractéristique 2) dans lequel une ou plusieurs branches retournent à une étape précédente. Les caractéristiques 6a, 6b et 6c correspondent aux options de représentation données respectivement dans les caractéristiques 2a, 2b et 2c. *Exemple:* (Caractéristique 6a illustrée) Une évolution, de S32 à S31, doit uniquement se produire si "c" est fausse et si "d" est vraie, c'est-à-dire que la séquence (S31, S32) sera répétée.

(suite à la page 190)

Table 46 *(continued)*

No.	Example	Rule
5a 5b 5c		*Sequence skip:* A "sequence skip" is a special case of sequence selection (Feature 2) in which one or more of the branches contain no steps. Features 5a, 5b, and 5c correspond to the representation options given in features 2a, 2b and 2c, respectively. *Example:* (Feature 5a shown) An evolution shall take place from S30 to S33 if "a" is false and "d" is true, that is, the sequence (S31, S32) will be skipped.
6a 6b 6c		*Sequence loop:* A "sequence loop" is a special case of sequence selection (Feature 2) in which one or more of the branches return to a preceding step. Features 6a, 6b and 6c correspond to the representation options given in features 2a, 2b and 2c, respectively. *Example:* (Feature 6a shown) An evolution shall take place from S32 to S31 if "c" is false and "d" is true, that is, the sequence (S31, S32) will be repeated.

(continued on page 191)

Tableau 46 *(fin)*

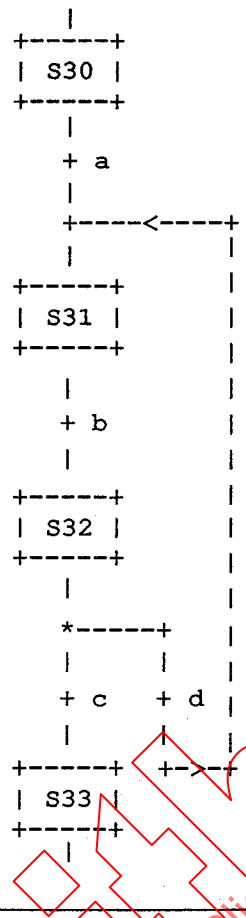
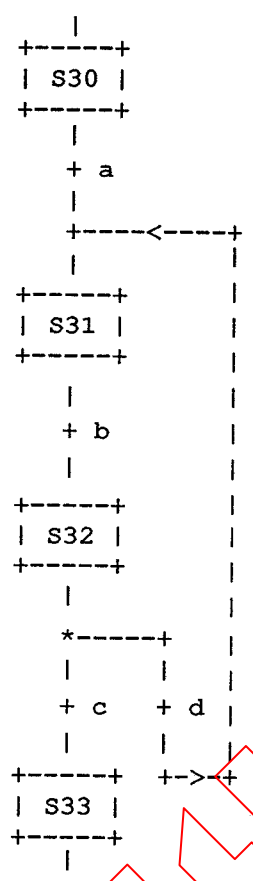
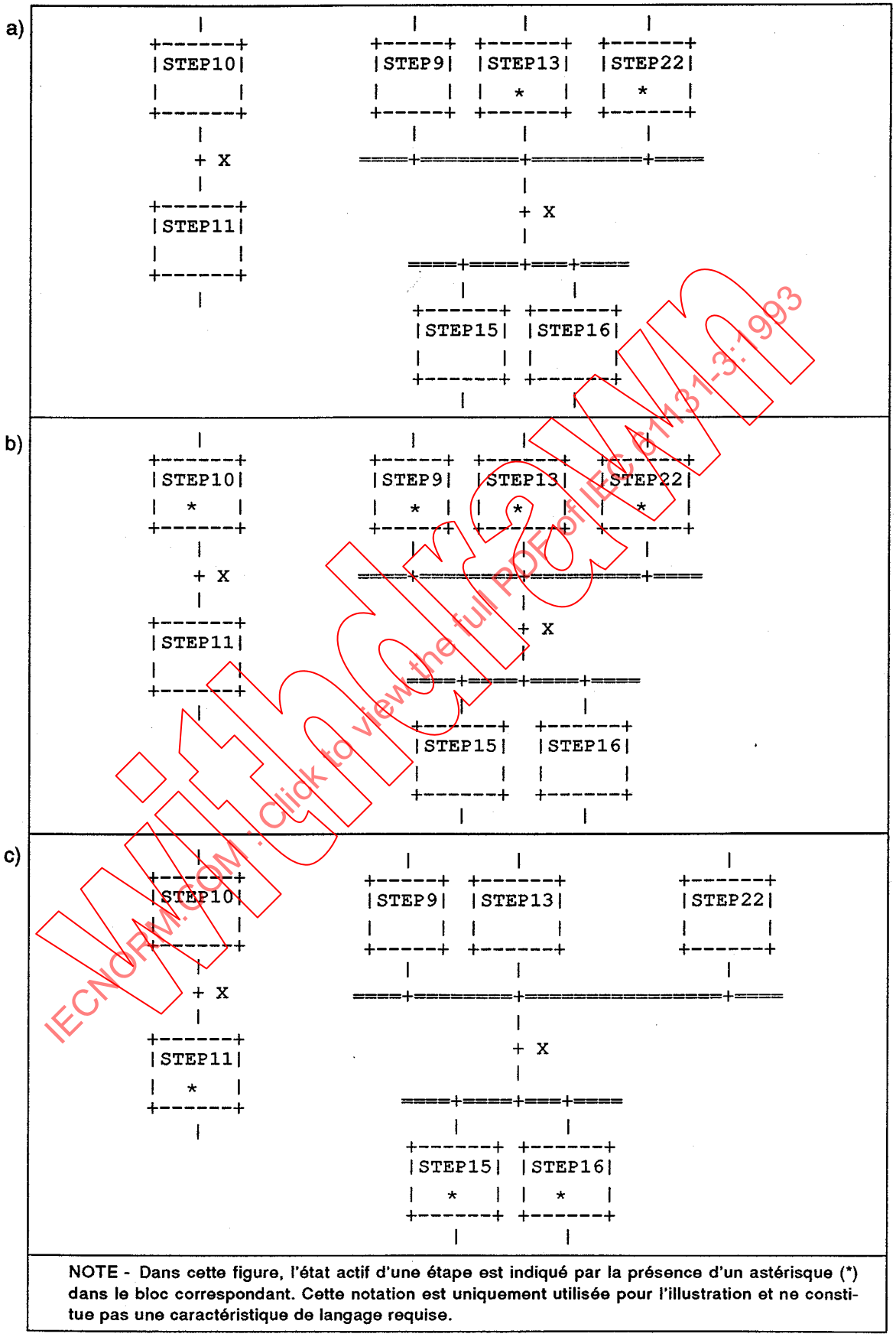
N°	Exemple	Règle
7		*Flèches directionnelles:* Lorsque cela est nécessaire pour plus de clarté, il est possible d'utiliser le caractère "inférieur à" (<) du jeu de caractères ISO/IEC 646 pour représenter un flux de commande de droite à gauche, ainsi que le caractère "supérieur à" (>) pour représenter un flux de commande de gauche à droite. Lorsque cette caractéristique est utilisée, le caractère correspondant doit se situer entre deux caractères "-", c'est-à-dire dans la séquence de caractères "-<-" ou "->-", comme cela est illustré dans l'exemple d'accompagnement.

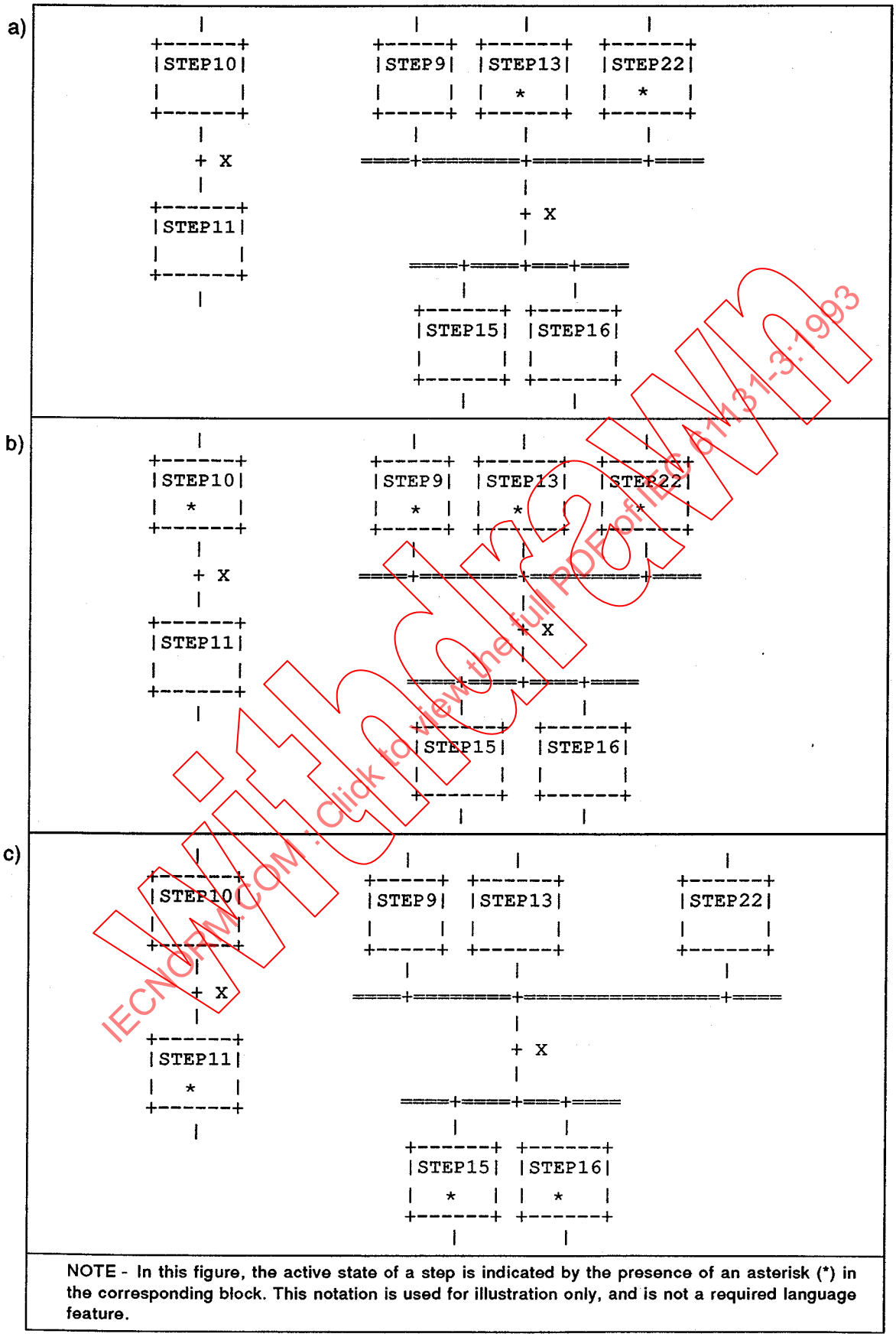
Table 46 *(concluded)*

No.	Example	Rule
7		*Directional arrows:* When necessary for clarity, the "less than" (<) character of the ISO/IEC 646 character set can be used to indicate right-to-left control flow, and the "greater than" (>) character to represent left-to-right control flow. When this feature is used, the corresponding character shall be located between two "-" characters, that is, in the character sequence "-<-" or "->-" as shown in the accompanying example.



NOTE - Dans cette figure, l'état actif d'une étape est indiqué par la présence d'un astérisque (*) dans le bloc correspondant. Cette notation est uniquement utilisée pour l'illustration et ne constitue pas une caractéristique de langage requise.

Figure 17 – Règles d'évolution de diagramme fonctionnel de séquence (SFC)
a) Transition non validée (X = indifférent)
b) Transition validée mais non effacée (X=0)
c) Transition effacée (X=1)



NOTE - In this figure, the active state of a step is indicated by the presence of an asterisk (*) in the corresponding block. This notation is used for illustration only, and is not a required language feature.

Figure 17 – SFC evolution rules
a) Transition not enabled (X = Don't care)
b) Transition enabled but not cleared (X=0)
c) Transition cleared (X=1)

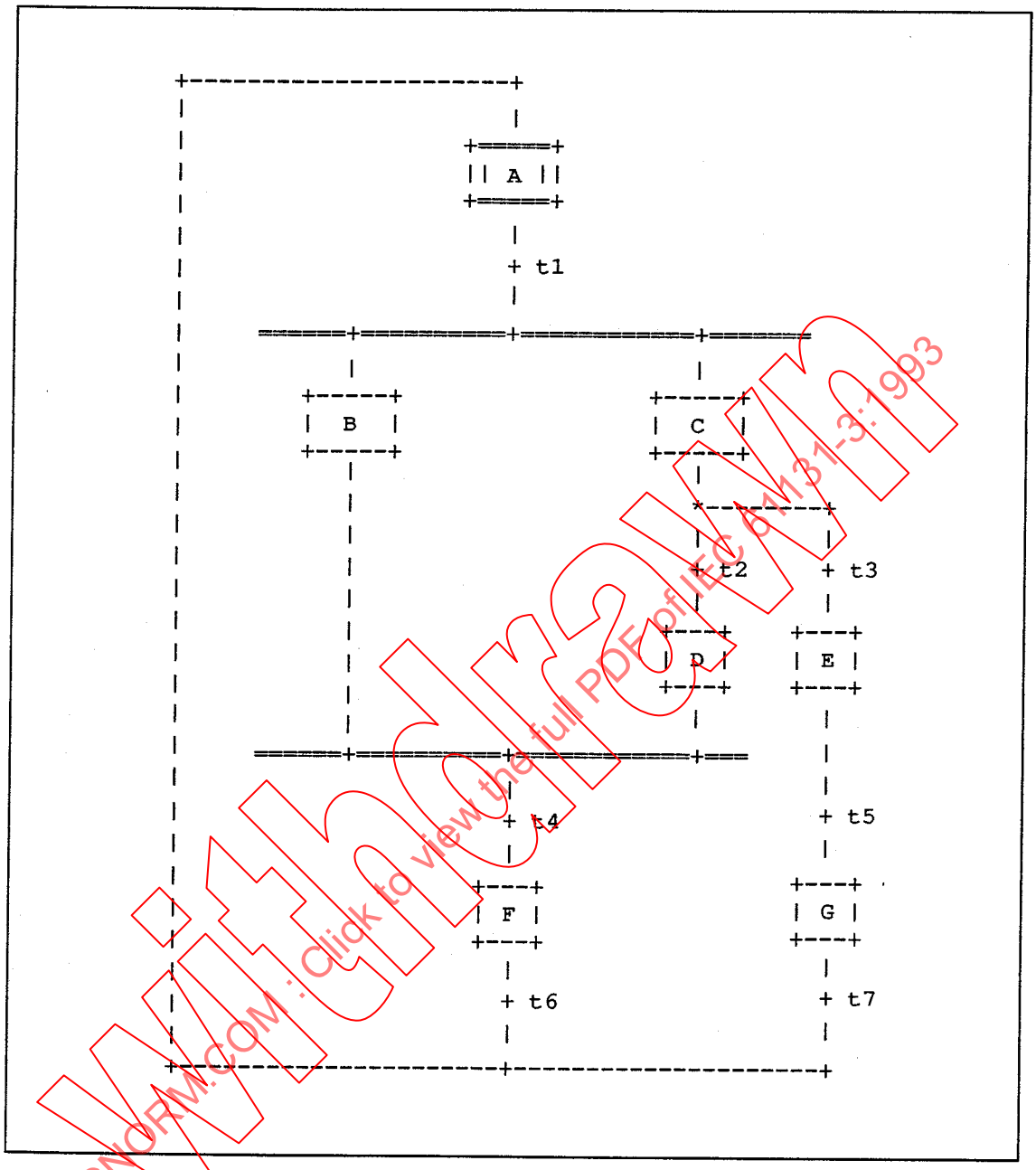


Figure 18a – Erreurs SFC: un élément de diagramme fonctionnel de séquence (SFC) "incertain" (voir 2.6.5)

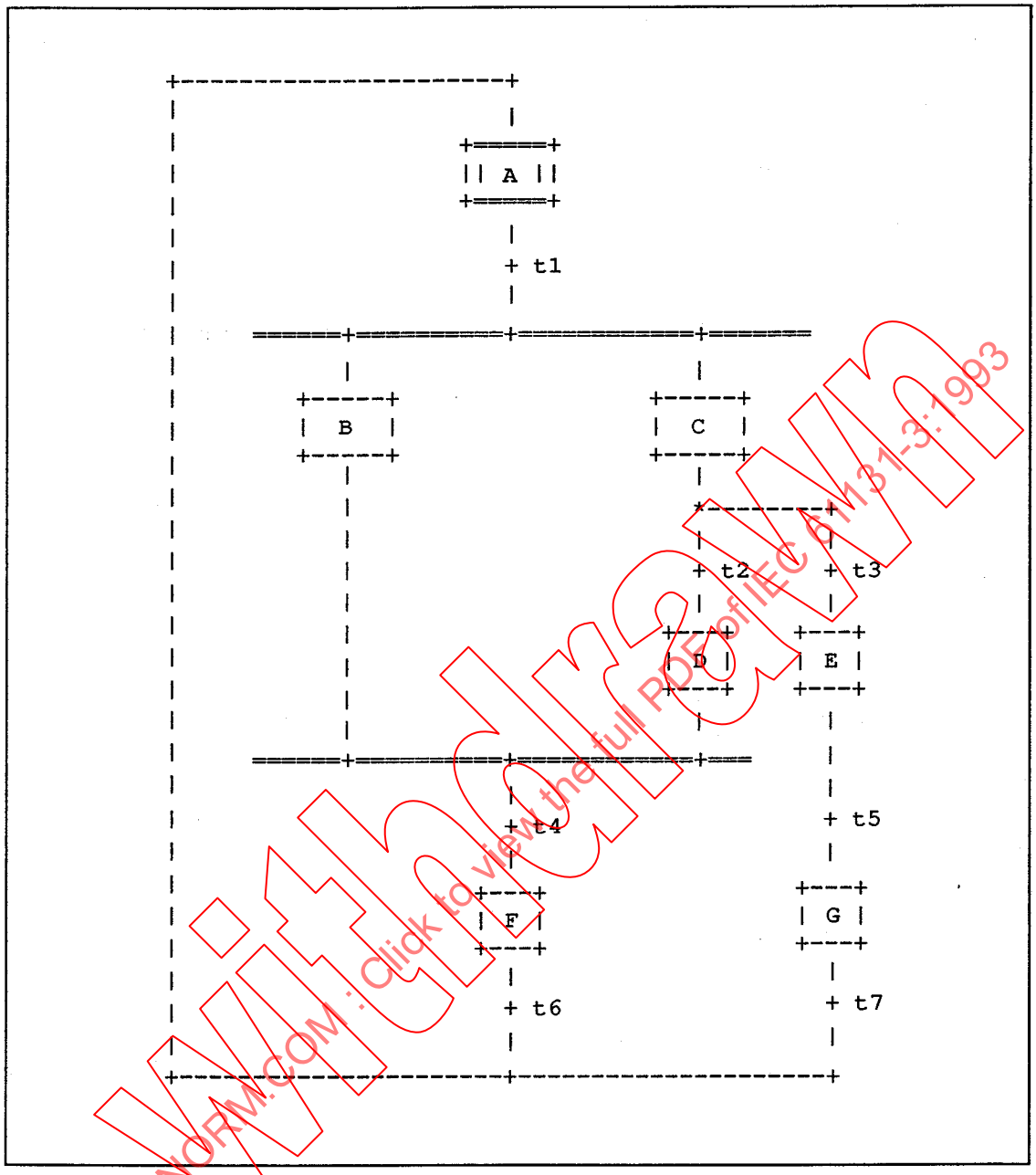


Figure 18a – SFC errors: an "unsafe" SFC (see 2.6.5)

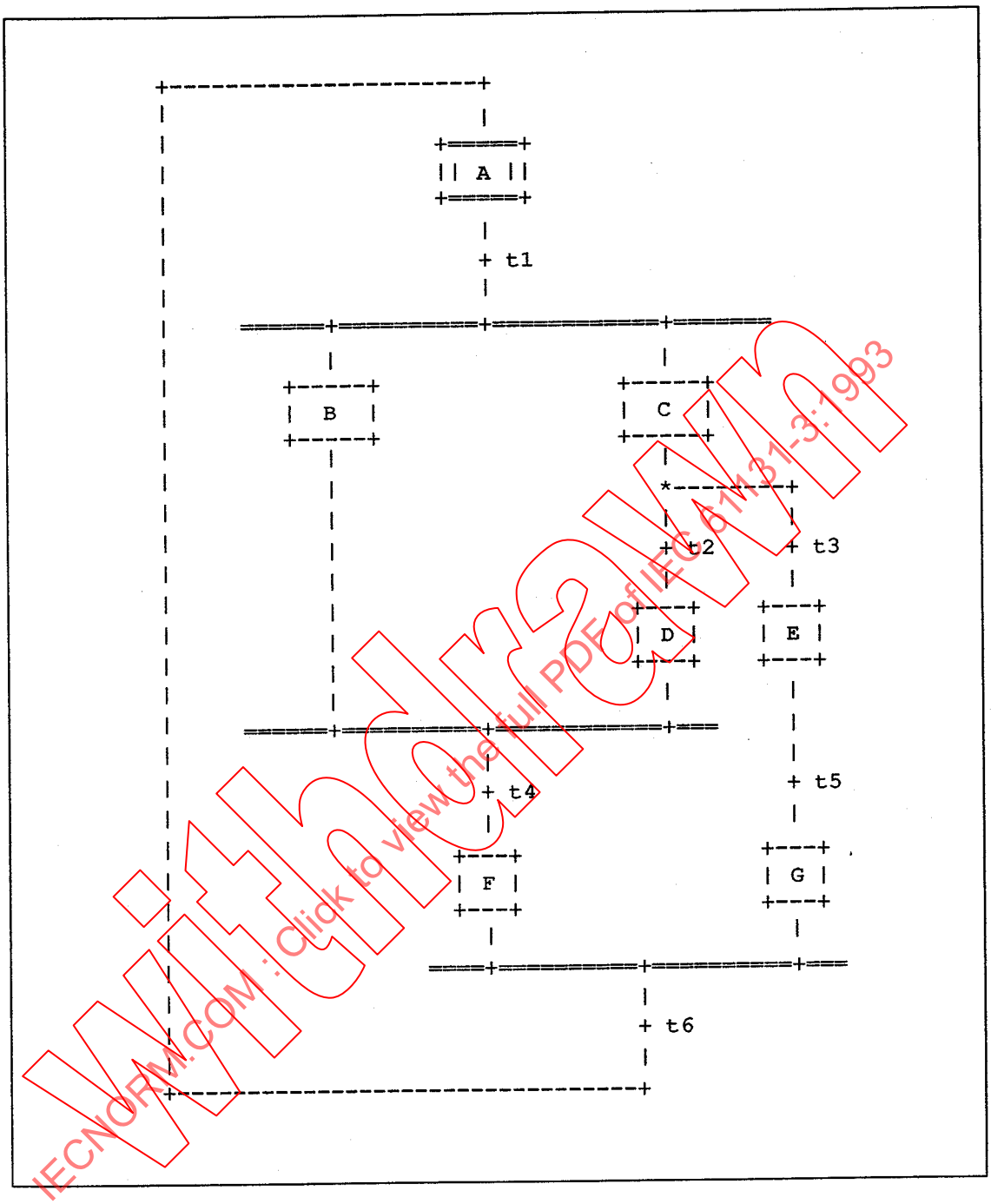


Figure 18b – Erreurs SFC: un élément de diagramme fonctionnel de séquence (SFC) "inaccessible" (voir 2.6.5)

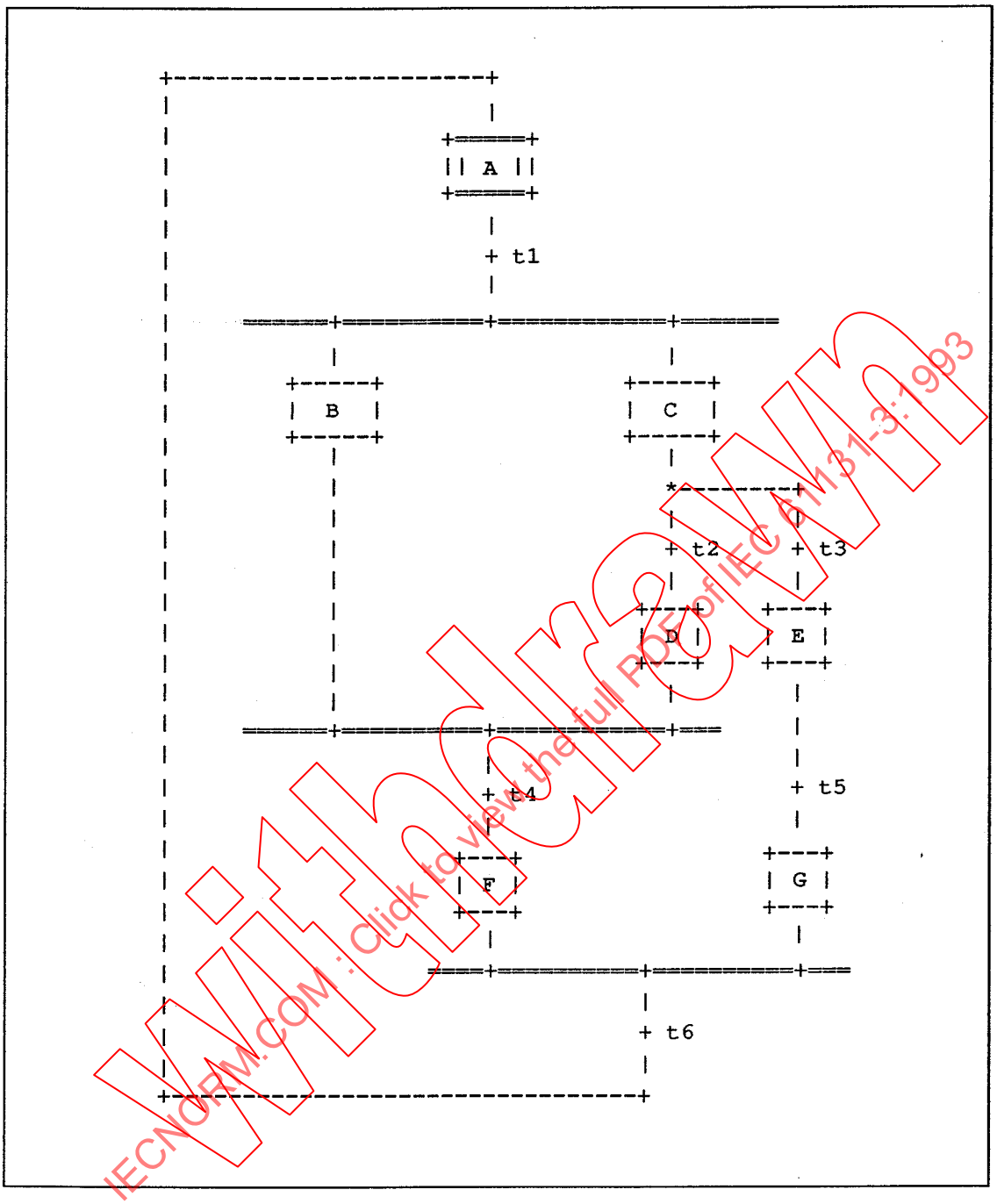


Figure 18b – SFC errors: an "unreachable" SFC (see 2.6.5)

2.6.6 *Compatibilité des éléments de diagramme fonctionnel en séquence (SFC)*

Les éléments de diagramme fonctionnel en séquence peuvent être représentés graphiquement ou littéralement, en utilisant les éléments définis plus haut. Pour plus de commodité, le tableau 47 résume les éléments qui sont mutuellement compatibles respectivement pour la représentation graphique et littérale.

Tableau 47 – Caractéristiques SFC compatibles

Tableau	Représentation graphique	Représentation littérale
40	1, 3a, 3b, 4	2, 3a, 4
41	1, 2, 3, 4, 4a, 4b, 7, 7a, 7b	5, 6, 7c, 7d
42	1, 2l, 2s, 2f	3s, 3i
43	1, 2, 4	3
44	1 à 9	–
45	1 à 10	1 à 10 (équivalent littéral)
46	1 à 7	1 à 6
57	Toutes	–

2.6.7 *Prescriptions en matière de compatibilité*

Afin de prétendre à la conformité aux prescriptions de 2.6, les éléments illustrés au tableau 48 doivent être acceptés et les prescriptions relatives à la compatibilité, définies en 2.6.6, doivent être satisfaites.

Tableau 48 – Prescriptions minimales relatives à la conformité des éléments de diagramme fonctionnel de séquence

Tableau	Représentation graphique	Représentation littérale
40	1	2
41	1 ou 2 ou 3 ou (4 et (4a ou 4b)) ou (7 et (7a ou 7b ou 7c ou 7 d))	5 ou 6
42	1 ou 2l ou 2f	3s ou 3i
43	1 ou 2 ou 4	3
45	1 ou 2	1 ou 2
46	1 et (2a ou 2b ou 2c) et 3 et 4	Identique (équivalent littéral)
57	(1 ou 2) et (3 ou 4) et (5 ou 6) et (7 ou 8) et (9 ou 10 et (11 ou 12)	Non requis

2.6.6 *Compatibility of SFC elements*

SFCS can be represented graphically or textually, utilizing the elements defined above. Table 47 summarizes for convenience those elements which are mutually compatible for graphical and textual representation, respectively.

Table 47 – Compatible SFC features

Table	Graphical representation	Textual representation
40	1, 3a, 3b, 4	2, 3a, 4
41	1, 2, 3, 4, 4a, 4b, 7, 7a, 7b	5, 6, 7c, 7d
42	1, 2l, 2s, 2f	3s, 3i
43	1, 2, 4	3
44	1 to 9	–
45	1 to 10	1 to 10 (textual equivalent)
46	1 to 7	1 to 6
57	All	–

2.6.7 *Compliance requirements*

In order to claim compliance with the requirements of 2.6, the elements shown in table 48 shall be supported and the compatibility requirements defined in 2.6.6 shall be observed.

Table 48 – SFC minimal compliance requirements

Table	Graphical representation	Textual representation
40	1	2
41	1 or 2 or 3 or (4 and (4a or 4b)) or (7 and (7a or 7b or 7c or 7 d))	5 or 6
42	1 or 2l or 2f	3s or 3i
43	1 or 2 or 4	3
45	1 or 2	1 or 2
46	1 and (2a or 2b or 2c) and 3 and 4	Same (textual equivalent)
57	(1 or 2) and (3 or 4) and (5 or 6) and (7 or 8) and (9 or 10 and (11 or 12)	Not required

2.7 *Eléments de configuration*

Comme cela est décrit en 1.4.1, une *configuration* se compose de *ressources,* de *tâches* (définies dans des *ressources),* de *variables globales* et de *chemins d'accès.* Chacun de ces éléments est défini, de façon détaillée, dans le présent paragraphe.

La figure 19a illustre un exemple graphique d'une configuration simple. Les déclarations paramétrables relatives aux blocs fonctionnels et aux programmes correspondants sont indiquées à la figure 19b. Cette figure sert de point de référence aux éléments des exemples de configuration donnés dans la partie restante du présent paragraphe comme dans la figure 20.

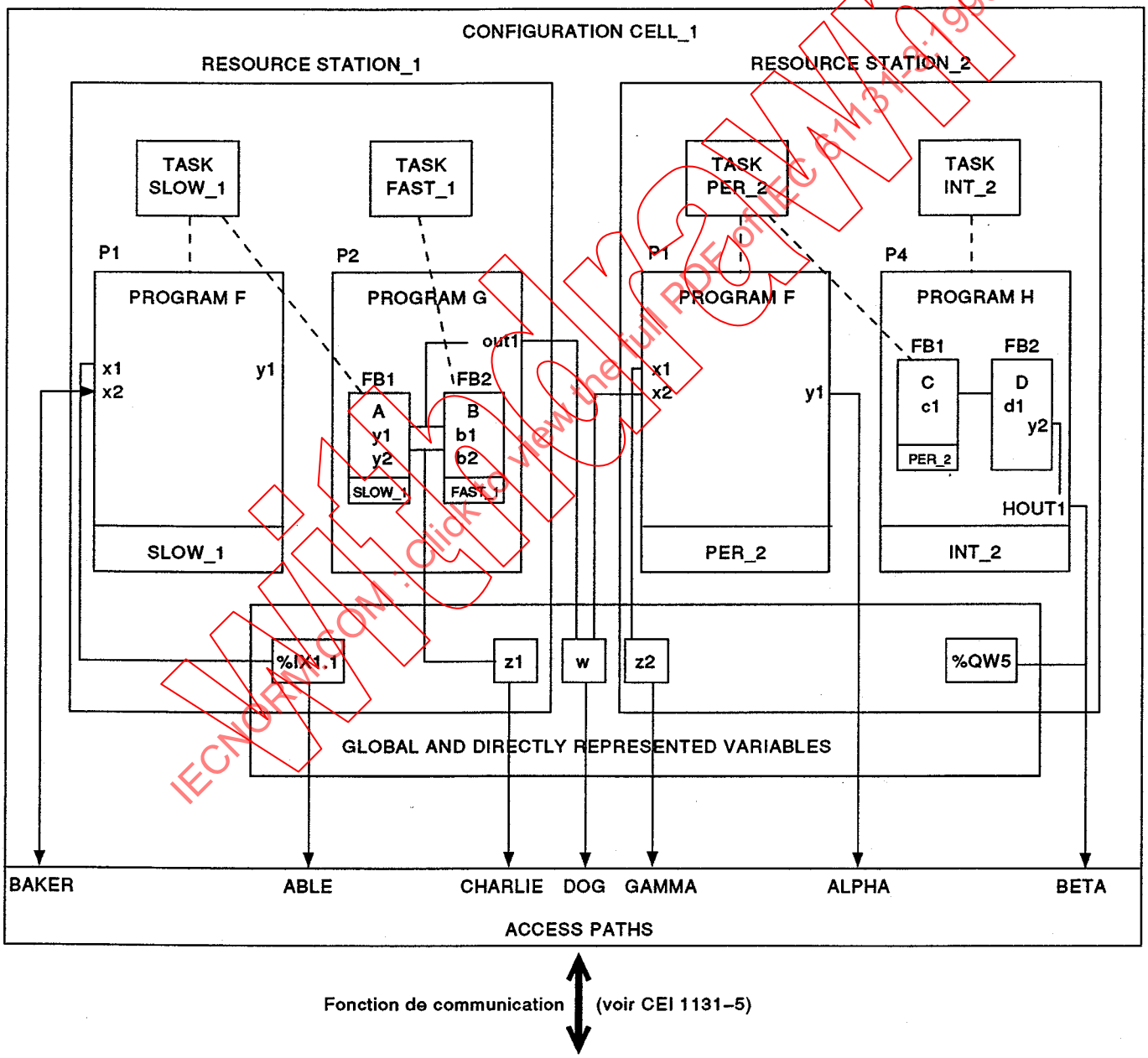


Figure 19a – Exemple graphique de configuration

2.7 *Configuration elements*

As described in 1.4.1, a *configuration* consists of *resources*, *tasks* (which are defined within *resources*), *global variables*, and *access paths*. Each of these elements is defined in detail in this subclause.

A graphic example of a simple configuration is shown in figure 19a. Skeleton declarations for the corresponding function blocks and programs are given in figure 19b. This figure serves as a reference point for the examples of configuration elements given in the remainder of this subclause such as in figure 20.

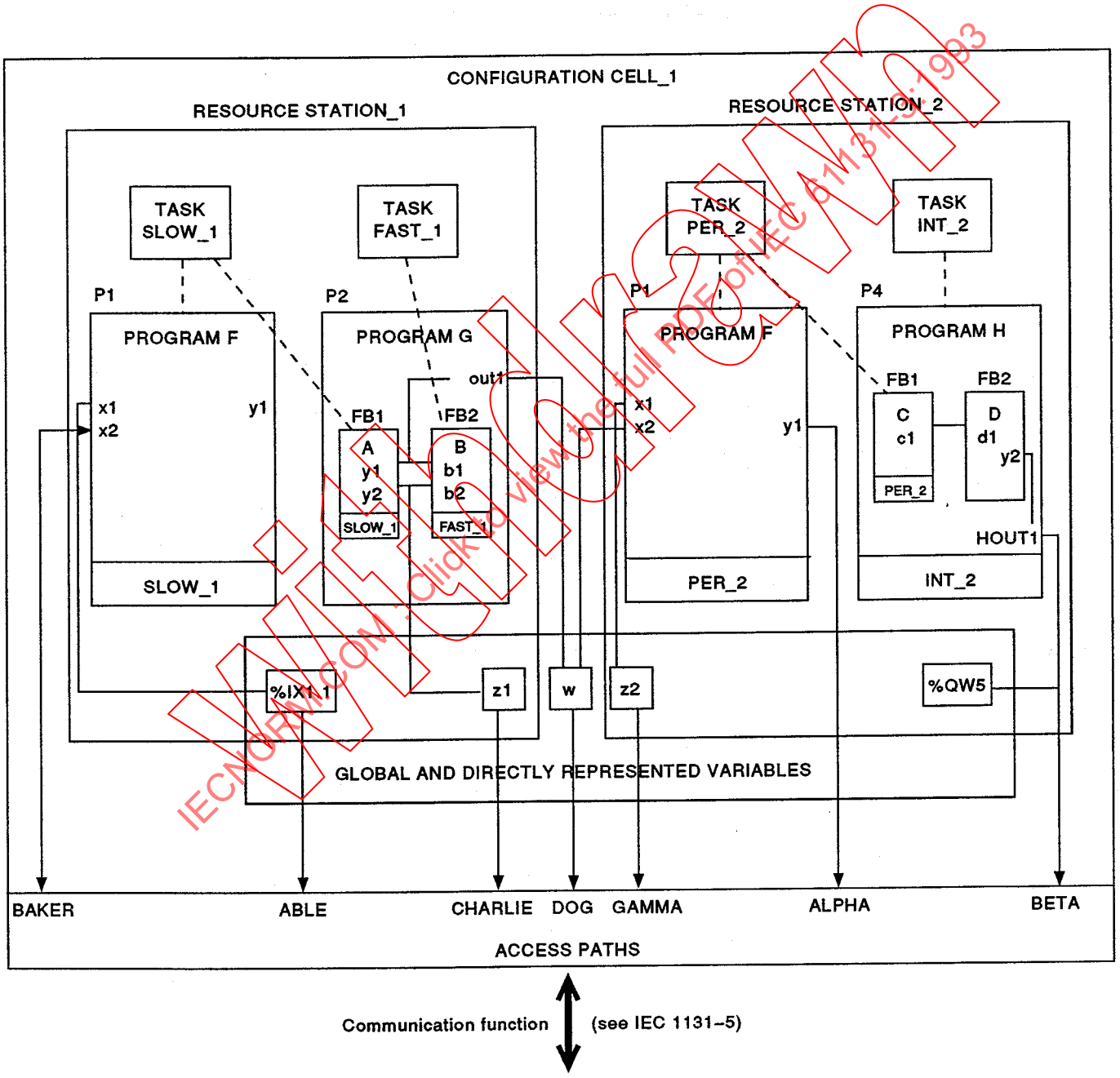


Figure 19a – Graphical example of a configuration

```
FUNCTION_BLOCK A
  VAR_OUTPUT  y1 : UINT ;
              y2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT b1 : UINT ;
            b2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK C
  VAR_OUTPUT c1 : BOOL ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK D
  VAR_INPUT d1 : BOOL ; END_VAR
  VAR_OUTPUT y2 : INT ; END_VAR
END_FUNCTION_BLOCK
```

```
PROGRAM F
  VAR_INPUT  x1 : BOOL ; x2 : UINT ; END_VAR
  VAR_OUTPUT y1 : BYTE ; END_VAR
END_PROGRAM
```

```
PROGRAM G
  VAR_OUTPUT out1 : UINT ; END_VAR
  VAR_EXTERNAL z1 : BYTE ; END_VAR
  VAR  FB1 : A ; FB2 : B ; END_VAR
FB1(...) ; out1 := FB1.y1 ; z1 := FB1.y2 ;
FB2(b1 := FB1.y1, b2 := FB1.y2) ;
END_PROGRAM
```

```
PROGRAM H
  VAR_OUTPUT HOUT1 : INT ; END_VAR
  VAR  FB1 : C ; FB2 : D ; END_VAR
FB1(...) ;
FB2(d1 := FB1.c1) ; HOUT1 := FB2.y2 ;
END_PROGRAM
```

Figure 19b – Déclarations de blocs fonctionnels et de programmes paramétrables relatives à un exemple de configuration

2.7.1 *Configurations, ressources et chemins d'accès*

Le tableau 49 énumère les caractéristiques de langage relatives à une déclaration de *configurations,* de *ressources,* de *variables globales,* et de *chemins d'accès.* L'énumération partielle de caractéristiques de déclarations TASK est également donnée; le paragraphe 2.7.2 fournit des informations supplémentaires concernant les *tâches.* La syntaxe formelle, pour ces caractéristiques, est donnée à l'annexe B.1.7. La figure 20 illustre des exemples de ces caractéristiques, correspondant à la configuration de l'exemple illustré à la figure 19b et aux déclarations d'acceptation illustrées à la figure 19a.

Le qualificatif ON, dans la construction RESOURCE...ON...END_RESOURCE est utilisé pour spécifier le type de "fonction de traitement" ainsi que ses fonctions "interface homme-machine" et "interface capteur et actionneur", sur lesquelles la *ressource* et ses *programmes* et *tâches* associés doivent être mis en oeuvre. Le fabricant doit fournir une "bibliothèque de ressource" de ces fonctions, comme l'illustre la figure 3. Pour l'utilisation dans la déclaration de la ressource, un identificateur (le *nom du type de ressource)* doit être associé à chaque élément dans cette "bibliothèque".

```
FUNCTION_BLOCK A
  VAR_OUTPUT  y1 : UINT ;
              y2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT b1 : UINT ;
            b2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK C
  VAR_OUTPUT c1 : BOOL ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK D
  VAR_INPUT d1 : BOOL ; END_VAR
  VAR_OUTPUT y2 : INT ; END_VAR
END_FUNCTION_BLOCK
```

```
PROGRAM F
  VAR_INPUT  x1 : BOOL ; x2 : UINT ; END_VAR
  VAR_OUTPUT y1 : BYTE ; END_VAR
END_PROGRAM
```

```
PROGRAM G
  VAR_OUTPUT out1 : UINT ; END_VAR
  VAR_EXTERNAL z1 : BYTE ; END_VAR
  VAR  FB1 : A ; FB2 : B ; END_VAR
FB1(...) ; out1 := FB1.y1 ; z1 := FB1.y2 ;
FB2(b1 := FB1.y1, b2 := FB1.y2) ;
END_PROGRAM
```

```
PROGRAM H
  VAR_OUTPUT HOUT1 : INT ; END_VAR
  VAR  FB1 : C ; FB2 : D ; END_VAR
FB1(...) ;
FB2(d1 := FB1.c1) ; HOUT1 := FB2.y2 ;
END_PROGRAM
```

Figure 19b – Skeleton function block and program declarations for configuration example

2.7.1 *Configurations, resources, and access paths*

Table 49 enumerates the language features for declaration of *configurations, resources, global* variables, and *access paths.* Partial enumeration of TASK declaration features is also given; additional information on *tasks* is provided in 2.7.2. The formal syntax for these features is given in B.1.7. Figure 20 provides examples of these features, corresponding to the example configuration shown in figure 19b and the supporting declarations in figure 19a.

The ON qualifier in the RESOURCE...ON...END_RESOURCE construction is used to specify the type of "processing function" and its "man-machine interface" and "sensor and actuator interface" functions upon which the *resource* and its associated *programs* and *tasks* are to be implemented. The manufacturer shall supply a *resource library* of such functions, as illustrated in figure 3. Associated with each element in this library shall be an identifier (the *resource type name*) for use in resource declaration.

Le *domaine* d'une déclaration VAR_GLOBAL doit se limiter à la *configuration* ou à la *ressource* dans laquelle il est déclaré, mis à part le fait qu'un *chemin d'accès* peut être déclaré à une *variable globale* dans une *ressource* utilisant la caractéristique 10d du tableau 49.

La construction VAR_ACCESS...END_VAR fournit un moyen permettant de spécifier des variables nommées auxquelles peuvent accéder certains des services de communication spécifiés dans la CEI 1131-5. Un *chemin d'accès* associe chacune de ces variables avec une variable d'entrée ou de sortie d'un *programme,* d'une *variable globale,* ou d'une *variable directement représentée,* telles que définies en 2.4.1.1. Si une telle variable est une *variable à plusieurs éléments* (*structure* ou *tableau),* un chemin d'accès peut être spécifié vers un élément de la variable. La direction du chemin d'accès peut être spécifiée comme READ_WRITE ou READ_ONLY, indiquant que les services de communication peuvent à la fois lire et modifier la valeur de la variable dans le premier cas, ou lire, mais non modifier, la valeur dans le second cas. Si aucune direction n'est spécifiée, la direction par défaut est READ_ONLY.

Tableau 49 – Caractéristiques de déclaration de ressource et de configuration

N°	Description
1	Construction CONFIGURATION...END_CONFIGURATION
2	Construction VAR_GLOBAL...END_VAR dans CONFIGURATION
3	Construction RESOURCE...ON...END_RESOURCE
4	Construction VAR_GLOBAL...END_VAR dans RESOURCE
5a	Construction TASK périodique dans RESOURCE (Note 1)
5b	Construction TASK non-périodique dans RESOURCE (Note 1)
6a	Déclaration PROGRAM avec association PROGRAM-to-TASK (Note 1)
6b	Déclaration PROGRAM avec association FUNCTION BLOCK-to-TASK (Note 1)
6c	Déclaration PROGRAM sans association TASK (Note 1)
7	Déclaration de variables directement représentées dans VAR_GLOBAL (Note 2)
8a	Connexion de variables directement représentées à des entrées PROGRAM
8b	Connexion de variables GLOBAL à des entrées PROGRAM
9a	Connexion de sorties PROGRAM à des variables directement représentées
9b	Connexion de sorties PROGRAM à des variables GLOBAL
10a	Construction VAR_ACCESS...END_VAR
10b	Chemins d'accès à des variables directement représentées
10c	Chemins d'accès à des entrées PROGRAM
10d	Chemins d'accès à des variables GLOBAL dans RESOURCES
10e	Chemins d'accès à des variables GLOBAL dans CONFIGURATION
10f	Chemins d'accès à des sorties PROGRAM

NOTES

1 Se reporter à 2.7.2, pour une description plus détaillée des caractéristiques TASK.

2 Se reporter à 2.4.3.1 pour une description plus détaillée des caractéristiques associées.

The *scope* of a VAR_GLOBAL declaration shall be limited to the *configuration* or resource in which it is declared, with the exception that an *access path* can be declared to a *global* variable in a *resource* using feature 10d in table 49.

The VAR_ACCESS...END_VAR construction provides a means of specifying named variables which can be accessed by some of the communication services specified in IEC 1131-5. An *access path* associates each such variable with an input or output variable of a *program*, a *global* variable, or a *directly represented* variable as defined in 2.4.1.1. If such a variable is a *multi-element variable* (*structure* or *array*), an access path can be specified to an element of the variable. The direction of the access path can be specified as READ_WRITE or READ_ONLY, indicating that the communication services can both read and modify the value of the variable in the first case, or read but not modify the value in the second case. If no direction is specified, the default direction is READ_ONLY.

Table 49 – Configuration and resource declaration features

No.	Description
1	CONFIGURATION...END_CONFIGURATION construction
2	VAR_GLOBAL...END_VAR construction within CONFIGURATION
3	RESOURCE...ON...END_RESOURCE construction
4	VAR_GLOBAL...END_VAR construction within RESOURCE
5a	Periodic TASK construction within RESOURCE (Note 1)
5b	Non-periodic TASK construction within RESOURCE (Note 1)
6a	PROGRAM declaration with PROGRAM-to-TASK association (Note 1)
6b	PROGRAM declaration with FUNCTION BLOCK-to-TASK association (Note 1)
6c	PROGRAM declaration with no TASK association (Note 1)
7	Declaration of directly represented variables in VAR_GLOBAL (Note 2)
8a	Connection of directly represented variables to PROGRAM inputs
8b	Connection of GLOBAL variables to PROGRAM inputs
9a	Connection of PROGRAM outputs to directly represented variables
9b	Connection of PROGRAM outputs to GLOBAL variables
10a	VAR_ACCESS...END_VAR construction
10b	Access paths to directly represented variables
10c	Access paths to PROGRAM inputs
10d	Access paths to GLOBAL variables in RESOURCES
10e	Access paths to GLOBAL variables in CONFIGURATION
10f	Access paths to PROGRAM outputs

NOTES

1 See 2.7.2 for further description of TASK features.

2 See 2.4.3.1 for further description of related features.

N°	Exemple
1	`CONFIGURATION CELL_1`
2	`  VAR_GLOBAL  w : UINT; END_VAR`
3	`  RESOURCE STATION_1 ON PROCESSOR_TYPE_1`
4	`    VAR_GLOBAL  z1 : BYTE ; END_VAR`
5a	`    TASK SLOW_1(INTEGRAL := t#20ms, PRIORITY := 2) ;`
5b	`    TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;`
6a	`    PROGRAM P1 WITH SLOW_1 :`
8a	`                F(x1 := %IX1.1) ;`
9b	`    PROGRAM P2 : G(out1 => w,`
6b	`                FB1 WITH SLOW_1,2`
6b	`                FB2 WITH FAST_1) ;`
3	`  END_RESOURCE`
3	`  RESOURCE STATION_2 ON PROCESSOR TYPE_2`
4	`    VAR_GLOBAL  z2 : BOOL ;`
7	`                AT %QW5 : INT ;`
4	`    END_VAR`
5a	`    TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;`
5b	`    TASK INT_2(SINGLE := z2, PRIORITY := 1) ;`
6a	`    PROGRAM P1 WITH PER_2 :`
8b	`                F(x1 := z2, x2 := w) ;`
6a	`    PROGRAM P4 WITH INT_2 :`
9a	`                H(HOUT1 => %QW5,`
6b	`                FB1 WITH  PER_2) ;`
3	`  END_RESOURCE`
10a	`  VAR_ACCESS`
10b	`    ABLE     : STATION_1.%IX1.1  : BOOL READ_ONLY ;`
10c	`    BAKER    : STATION_1.P1.x2   : UINT READ_ONLY ;`
10d	`    CHARLIE  : STATION_1.z1      : BYTE ;`
10e	`    DOG      : w                 : UINT READ_ONLY ;`
10f	`    ALPHA    : STATION_2.P1.y1   : BYTE READ_ONLY ;`
10f	`    BETA     : STATION_2.P4.HOUT1 : INT READ_ONLY ;`
10d	`    GAMMA    : STATION_2.z2      : BOOL READ_WRITE ;`
10a	`  END_VAR`
1	`END_CONFIGURATION`

NOTES

1 La représentation graphique et semi-graphique de ces caractéristiques est tolérée, mais elle n'est pas couverte par le domaine d'application de la présente partie de la CEI 1131.

2 Si le type de donnée déclarée pour une variable dans une déclaration de VAR_ACCESS est différent du type de donnée déclaré ailleurs pour la même variable, par exemple si, dans l'exemple ci-dessus, la variable BAKER est déclarée du type WORD, cela doit être considéré comme une ERREUR.

Figure 20 – Exemples de caractéristiques de déclarations CONFIGURATION et RESOURCE

2.7.2 *Tâches*

Aux fins de la présente partie de la CEI 1131, une *tâche* se définit comme un élément de commande d'exécution, capable de lancer, soit de façon périodique, soit lors de l'apparition du front montant d'une variable booléenne spécifiée, l'exécution d'un ensemble d'unités d'organisation de programmes, qui peuvent inclure des *programmes* et des *blocs fonctionnels* dont les instances sont spécifiées dans la déclaration de *programmes.*

No.	Example
1	CONFIGURATION CELL_1
2	VAR_GLOBAL w : UINT; END_VAR
3	RESOURCE STATION_1 ON PROCESSOR_TYPE_1
4	VAR_GLOBAL z1 : BYTE ; END_VAR
5a	TASK SLOW_1(INTEGRAL := t#20ms, PRIORITY := 2) ;
5b	TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
6a	PROGRAM P1 WITH SLOW_1 :
8a	F(x1 := %IX1.1) ;
9b	PROGRAM P2 : G(out1 => w,
6b	FB1 WITH SLOW_1,
6b	FB2 WITH FAST_1) ;
3	END_RESOURCE
3	RESOURCE STATION_2 ON PROCESSOR TYPE_2
4	VAR_GLOBAL z2 : BOOL ;
7	AT %QW5 : INT ;
4	END_VAR
5a	TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;
5b	TASK INT_2(SINGLE := z2, PRIORITY := 1) ;
6a	PROGRAM P1 WITH PER_2 :
8b	F(x1 := z2, x2 := w) ;
6a	PROGRAM P4 WITH INT_2 :
9a	H(HOUT1 => %QW5,
6b	FB1 WITH PER_2) ;
3	END_RESOURCE
10a	VAR_ACCESS
10b	ABLE : STATION_1.%IX1.1 : BOOL READ_ONLY ;
10c	BAKER : STATION_1.P1.x2 : UINT READ_ONLY ;
10d	CHARLIE : STATION_1.z1 : BYTE ;
10e	DOG : w : UINT READ_ONLY ;
10f	ALPHA : STATION_2.P1.y1 : BYTE READ_ONLY ;
10f	BETA : STATION_2.P4.HOUT1 : INT READ_ONLY ;
10d	GAMMA : STATION_2.z2 : BOOL READ_WRITE ;
10a	END_VAR
1	END_CONFIGURATION

NOTES

1 Graphical and semigraphic representation of these features is allowed but is beyond the scope of this part of IEC 1131.

2 It is an error if the data type declared for a variable in a VAR_ACCESS statement is not the same as the data type declared for the variable elsewhere, e.g., if variable BAKER is declared of type WORD in the above examples.

Figure 20 – Examples of CONFIGURATION and RESOURCE declaration features

2.7.2 *Tasks*

For the purposes of IEC 1131-3, a *task* is defined as an execution control element which is capable of invoking, either on a periodic basis or upon the occurrence of the rising edge of a specified Boolean variable, the execution of a set of program organization units, which can include *programs* and *function blocks* whose instances are specified in the declaration of *programs*.

Les tâches et leur association au sein d'unités d'organisation de programmes peuvent être représentées graphiquement ou littéralement, comme l'indique le tableau 50, en tant que parties de *ressources* dans des *configurations.* Une tâche est implicitement validée ou invalidée par sa ressource associée, conformément aux mécanismes définis en 1.4.1. La commande des unités d'organisation de programmes par des tâches validées doit être conforme aux règles suivantes:

1) Les unités d'organisation de programmes associées doivent être prévues pour l'exécution ordonnancée lors de l'apparition de chaque front montant de l'entrée SINGLE de la tâche.

2) Si l'entrée INTERVAL est différente de zéro, les unités d'organisation de programmes associées doivent être programmées pour une exécution périodique à l'intervalle spécifié, tant que l'entrée SINGLE reste à zéro (0). Si l'entrée INTERVAL est à zéro (la valeur par défaut), aucun ordonnancement périodique des unités d'organisation de programmes associées ne doit avoir lieu.

3) L'entrée PRIORITY d'une tâche établit la priorité d'ordonnancement des unités d'organisation de programmes associées; zéro (0) étant la priorité la plus élevée et les priorités décroissantes successives ayant successivement des valeurs numériques plus élevées. Comme l'indique le tableau 50, la priorité d'une unité d'organisation de programme (c'est-à-dire, la priorité de sa tâche associée) peut être utilisée pour un ordonnancement *préemptif* ou *non-préemptif.*

a) dans un ordonnancement *non préemptif,* la puissance de traitement devient disponible sur une *ressource* lorsque l'exécution d'une unité d'organisation de programme ou d'une fonction du système d'exploitation est complète. Lorsque la puissance de traitement est disponible, l'unité d'organisation de programme, ayant la priorité la plus élevée doit commencer l'exécution. Si plus d'une unité d'organisation de programme est en attente au niveau de priorité le plus élevé, alors l'unité d'exécution de programme ayant le temps d'attente le plus élevé, à la priorité la plus élevée, doit être exécutée.

b) Dans un ordonnancement *préemptif,* lorsqu'une unité d'exécution de programme est programmée, celle-ci peut *interrompre* l'exécution d'une unité d'organisation de programme ayant un niveau de priorité inférieur sur la même *ressource,* c'est-à-dire que l'exécution de l'unité de priorité inférieure peut être interrompue jusqu'à la fin de l'exécution de l'unité ayant le niveau de priorité le plus élevé. Une unité d'exécution de programme ne doit pas interrompre l'exécution d'une autre unité ayant le même niveau de priorité.

NOTE - En fonction des priorités prévues, une unité d'exécution de programme peut ne pas commencer l'exécution au moment où elle est programmée. Cependant, dans l'exemple illustré au tableau 50, toutes les unités d'organisation de programmes se conforment à leur *délais,* c'est-à-dire qu'elles sont toutes exécutées avant l'instant prévu pour une nouvelle exécution. Le fabricant doit fournir des informations pour permettre à l'utilisateur de déterminer si tous les *délais* seront satisfaits dans une configuration proposée.

4) Un *programme* sans association de tâche doit avoir le niveau de priorité le plus bas du système. Tout programme de ce type doit être prévu pour l'exécution dès le "démarrage" de sa *ressource,* telle que définie au paragraphe 1.4.1, et doit être prévu pour être à nouveau exécuté dès la fin de son exécution.

5) Lorsqu'une *instance de bloc fonctionnel* est associée à une tâche, son exécution doit se faire sous la commande exclusive de la tâche, indépendamment des règles d'évaluation de l'unité d'organisation de programme dans laquelle est déclarée l'instance de bloc fonctionnel associé à une tâche.

Tasks and their association with program organization units can be represented graphically or textually, as shown in table 50, as part of *resources* within *configurations*. A task is implicitly enabled or disabled by its associated resource according to the mechanisms defined in 1.4.1. The control of program organization units by enabled tasks shall conform to the following rules:

1) The associated program organization units shall be scheduled for execution upon each rising edge of the SINGLE input of the task.

2) If the INTERVAL input is non-zero, the associated program organization units shall be scheduled for execution periodically at the specified interval as long as the SINGLE input stands at zero (0). If the INTERVAL input is zero (the default value), no periodic scheduling of the associated program organization units shall occur.

3) The PRIORITY input of a task establishes the scheduling priority of the associated program organization units, with zero (0) being highest priority and successively lower priorities having successively higher numeric values. As shown in table 50, the priority of a program organization unit (that is, the priority of its associated task) can be used for *preemptive* or *non-preemptive* scheduling.

a) In *non-preemptive* scheduling, processing power becomes available on a *resource* when execution of a program organization unit or operating system function is complete. When processing power is available, the program organization unit with highest scheduled priority shall begin execution. If more than one program organization unit is waiting at the highest scheduled priority, then the program organization unit with the longest waiting time at the highest scheduled priority shall be executed.

b) In *preemptive* scheduling, when a program organization unit is scheduled, it can *interrupt* the execution of a program organization unit of lower priority on the same *resource*, that is, the execution of the lower-priority unit can be suspended until the execution of the higher-priority unit is completed. A program organization unit shall not interrupt the execution of another unit of the same or higher priority.

NOTE - Depending on schedule priorities, a program organization unit might not begin execution at the instant it is scheduled. However, in the examples shown in table 50, all program organization units meet their *deadlines*, that is, they all complete execution before being scheduled for re-execution. The manufacturer shall provide information to enable the user to determine whether all deadlines will be met in a proposed configuration.

4) A *program* with no task association shall have the lowest system priority. Any such program shall be scheduled for execution upon "starting" of its *resource*, as defined in 1.4.1, and shall be re-scheduled for execution as soon as its execution terminates.

5) When a *function block instance* is associated with a task, its execution shall be under the exclusive control of the task, independent of the rules of evaluation of the program organization unit in which the task-associated function block instance is declared.

6) L'exécution d'une *instance de bloc fonctionnel,* qui n'est pas directement associée à une tâche, doit suivre les règles normales concernant l'ordre d'évaluation des éléments de langage relatifs à l'unité d'organisation de programme (qui peut elle-même être sous la commande d'une tâche) dans laquelle l'instance de bloc fonctionnel est déclarée.

7) L'exécution de blocs fonctionnels dans un programme doit être synchronisée, afin d'assurer que la concurrence d'accès des données est effectuée conformément aux règles suivantes:

a) Si un bloc fonctionnel reçoit plus d'une entrée en provenance d'un autre bloc fonctionnel, alors, lorsque le premier bloc fonctionnel est exécuté, toutes les entrées en provenance de l'autre bloc fonctionnel doivent représenter les résultats de la même évaluation. Par exemple, dans l'exemple illustré par la figure 21a, lorsque Y2 est évaluée, les entrées Y2.A et Y2.B doivent représenter les sorties Y1.C et Y1.D, à partir de la même évaluation de Y1 (et non de deux évaluations différentes).

b) Si deux ou plusieurs blocs fonctionnels reçoivent des entrées en provenance du même bloc fonctionnel, et si les blocs de "destination" sont tous explicitement ou implicitement associés à la même tâche, alors les entrées de tous ces blocs de "destination" doivent représenter, au moment de leur évaluation, les résultats de la même évaluation du bloc "source". Par exemple, dans l'exemple illustré par les figures 21b et 21c, lorsque Y2 et Y3 sont évalués au cours de l'évaluation du programme P1, les entrées Y2.A et Y2.B doivent être les résultats de la même évaluation de Y1 que celle des entrées Y3.A et Y3.B.

Il est nécessaire de prévoir une zone de mémorisation pour les sorties de fonctions ou de blocs fonctionnels qui sont explicitement associées à des tâches, ou qui sont utilisées, en tant qu'entrées pour les unités d'organisation de programmes qui ont des associations de tâches explicites, selon le cas nécessaire pour se conformer aux règles indiquées ci-dessus.

6) Execution of *a function block instance* which is not directly associated with a task shall follow the normal rules for the order of evaluation of language elements for the program organization unit (which can itself be under the control of a task) in which the function block instance is declared.

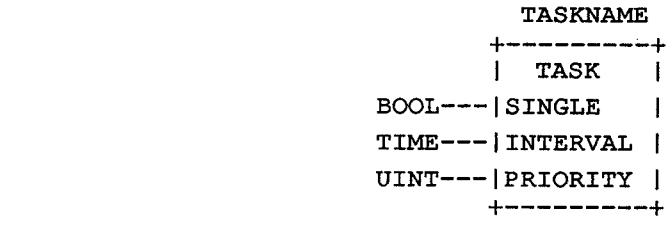
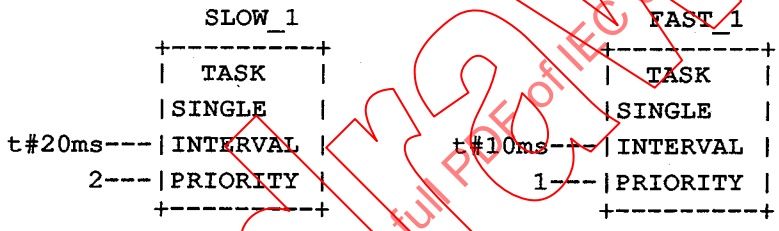
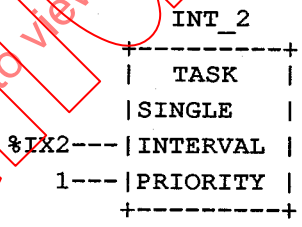
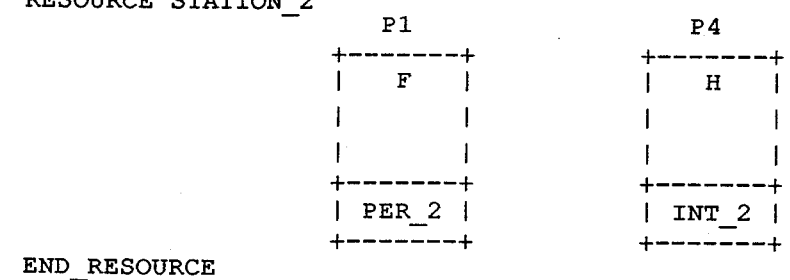
7) The execution of function blocks within a program shall be synchronized to ensure that data concurrency is achieved according to the following rules:

a) If a function block receives more than one input from another function block, then when the former is executed, all inputs from the latter shall represent the results of the same evaluation. For instance, in the example represented by figure 21a, when Y2 is evaluated, the inputs Y2.A and Y2.B shall represent the outputs Y1.C and Y1.D from the same (not two different) evaluations of Y1.

b) If two or more function blocks receive inputs from the same function block, and if the "destination" blocks are all explicitly or implicitly associated with the same task, then the inputs to all such "destination" blocks at the time of their evaluation shall represent the results of the same evaluation of the "source" block. For instance, in the example represented by figures 21b and 21c, when Y2 and Y3 are evaluated in the normal course of evaluating program P1, the inputs Y2.A and Y2.B shall be the results of the same evaluation of Y1 as the inputs Y3.A and Y3.B.

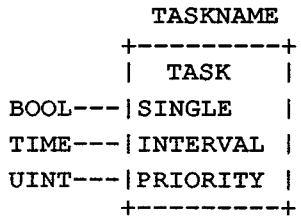
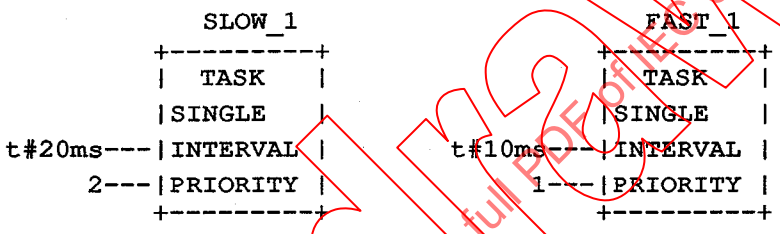
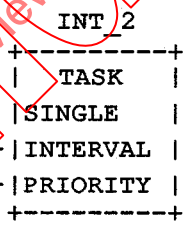
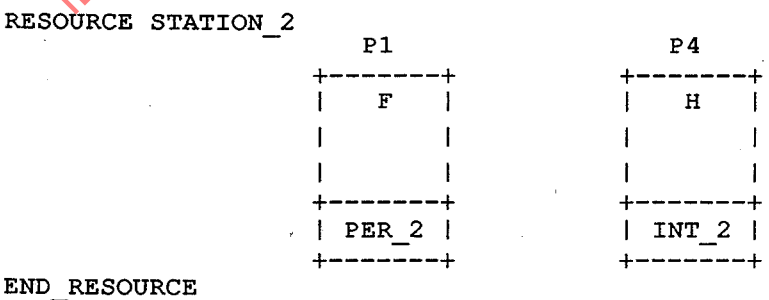
Provision shall be made for storage of the outputs of functions or function blocks which have explicit task associations, or which are used as inputs to program organization units which have explicit task associations, as necessary to satisfy the rules given above.

Tableau 50 – Caractéristiques de tâche

N°	Description/Exemples
1a	Déclaration littérale de TACHE périodique (cactéristique 5a du tableau 49)
1b	Déclaration littérale de TACHE non périodique (caractéristique 5b du tableau 49)
	Représentation graphique de TACHEs dans une RESOURCE
	
2a	Représentation graphique de TACHES périodiques
	
2b	Représentation graphique de TACHES non-périodiques
	
3a	Association littérale avec des PROGRAMS (caractéristique 6a du tableau 49)
3b	Association littérale avec des FUNCTION BLOCKS (caractéristique 6b du tableau 49)
4a	Association graphique avec des PROGRAMS (dans RESOURCES)
	

(suite à la page 214)

Table 50 – Task features

No.	Description/Examples
1a	Textual declaration of periodic TASK (feature 5a of table 49)
1b	Textual declaration of non-periodic TASK (feature 5b of table 49)
	Graphical representation of TASKs within a RESOURCE
	
2a	Graphical representation of periodic TASKS
	
2b	Graphical representation of non-periodic TASK
	
3a	Textual association with PROGRAMS (feature 6a of table 49)
3b	Textual association with FUNCTION BLOCKs (feature 6b of table 49)
4a	Graphical association with PROGRAMS (within RESOURCES)
	

(continued on page 215)

Tableau 50 *(suite)*

N°	Description/Exemples
4b	Association graphique avec des FUNCTION BLOCKS (dans des PROGRAMS à l'intérieur de RESOURCES)

```
RESOURCE STATION_1
                          P2
         +-------------------------------------------------+
         |                G                                |
         |                                                 |
         |                                                 |
         |         FB1                   FB2               |
         |      +-------+             +-------+            |
         |      |   A   |             |   B   |            |
         |      |       |             |       |            |
         |      |       |             |       |            |
         |      +-------+             +-------+            |
         |      |SLOW_1 |             |FAST_1 |            |
         |      +-------+             +-------+            |
         +-------------------------------------------------+
END_RESOURCE
```

N°	Description/Exemples
5a	Ordonnancement non préemptif

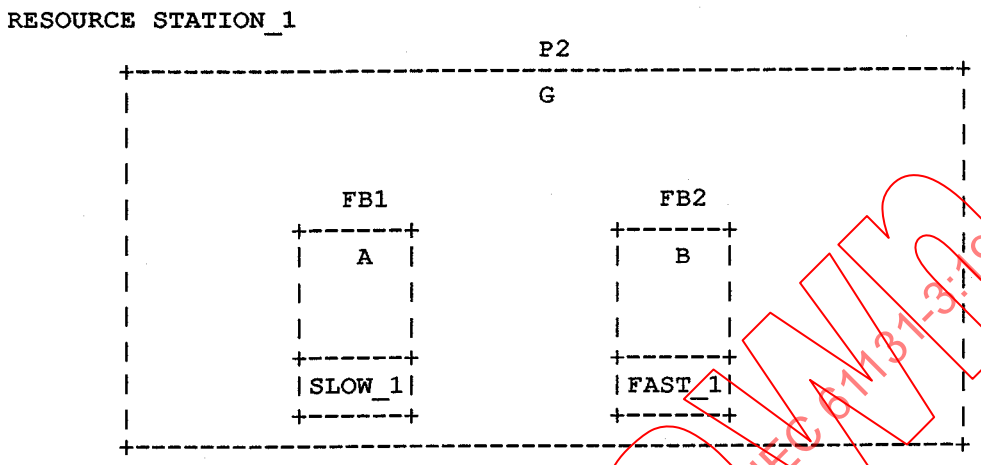
Exemple 1:
– RESOURCE STATION_1 telle que configurée à la figure 20
– Temps d'exécution: P1 = 2 ms, P2 = 8 ms
P2.FB1 = P2.FB2 = 2 ms (note 3)
– STATION_1 démarre à t = 0

Calendrier (se répète toutes les 40 ms)

t(ms)	En cours d'exécution	En attente
0	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2
2	P1 @ 2	P2.FB1 @ 2, P2
4	P2.FB1 @ 2	P2
6	P2	
10	P2	P2.FB2 @ 1
14	P2.FB2 @ 1	P2
16	P2	(P2 relancé)
20	P2	P2.FB.2 @ 1, P1 @ 2, P2.FB1 @ 2
24	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2
26	P1 @ 2	P2.FB1 @ 2, P2
28	P2.FB1 @ 2	P2
30	P2.FB2 @ 1	P2
32	P2	
40	P2.FB2 @ 1	P1 @ 2, P2.FB1 @ 2, P2

(suite à la page 216)

Table 50 *(continued)*

<table>
<tr><th>No.</th><th>Description/Examples</th></tr>
<tr><td>4b</td><td>Graphical association with FUNCTION BLOCKS
(within PROGRAMS inside RESOURCES)</td></tr>
<tr><td></td><td>


</td></tr>
<tr><td>5a</td><td>Non-preemptive scheduling

Example 1:
– RESOURCE STATION_1 as configured in figure 20
– Execution times: P1 = 2 ms, P2 = 8 ms
P2.FB1 = P2.FB2 = 2 ms (note 3)
– STATION_1 starts at t = 0

Schedule (repeats every 40 ms)
<table>
<tr><th>t(ms)</th><th>Executing</th><th>Waiting</th></tr>
<tr><td>0</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td>2</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td>4</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td>6</td><td>P2</td><td></td></tr>
<tr><td>10</td><td>P2</td><td>P2.FB2 @ 1</td></tr>
<tr><td>14</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
<tr><td>16</td><td>P2</td><td>(P2 restarts)</td></tr>
<tr><td>20</td><td>P2</td><td>P2.FB.2 @ 1, P1 @ 2, P2.FB1 @ 2</td></tr>
<tr><td>24</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td>26</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td>28</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td>30</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
<tr><td>32</td><td>P2</td><td></td></tr>
<tr><td>40</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
</table>
</td></tr>
</table>

(continued on page 217)

Tableau 50 *(suite)*

<table>
<tr><td>N°</td><td colspan="3">Description/Exemples</td></tr>
<tr><td></td><td colspan="3">Exemple 2:
– RESOURCE STATION_2 telle que configurée à la figure 20
– Temps d'exécution: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (note 4)
– INT_2 est déclenchée à t = 25, 50, 90, ... ms
– STATION_2 démarre à t = 0</td></tr>
<tr><td></td><td colspan="3">Calendrier</td></tr>
<tr><td></td><td>t(ms)</td><td>En cours d'exécution</td><td>En attente</td></tr>
<tr><td></td><td>0</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>25</td><td>P1 @ 2</td><td>P4.FB1 @ 2, P4 @ 1</td></tr>
<tr><td></td><td>30</td><td>P4 @ 1</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>35</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td></td><td>50</td><td>P4 @ 1</td><td>P1 @ 2, P4.FB1 @ 2</td></tr>
<tr><td></td><td>55</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>85</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td></td><td>90</td><td>P4.FB1 @ 2</td><td>P4 @ 1</td></tr>
<tr><td></td><td>95</td><td>P4 @ 1</td><td></td></tr>
<tr><td></td><td>100</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>5b</td><td colspan="3">Ordonnancement préemptif</td></tr>
<tr><td></td><td colspan="3">Exemple 3:
– RESOURCE STATION_1 telle que configurée à la figure 20
– Temps d'exécution: P1 = 2 ms, P2 = 8 ms; P2.FB1 = P2.FB2 = 2 ms (note 3)
– STATION_1 démarre à t = 0</td></tr>
<tr><td></td><td colspan="3">Calendrier</td></tr>
<tr><td></td><td>t(ms)</td><td>En cours d'exécution</td><td>En attente</td></tr>
<tr><td></td><td>0</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td></td><td>2</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td></td><td>4</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td></td><td>6</td><td>P2</td><td></td></tr>
<tr><td></td><td>10</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
<tr><td></td><td>12</td><td>P2</td><td></td></tr>
<tr><td></td><td>16</td><td>P2</td><td>(P2 relancé)</td></tr>
<tr><td></td><td>20</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
</table>

(suite à la page 218)

Table 50 *(continued)*

<table>
<tr><td>No.</td><td colspan="3">Description/Examples</td></tr>
<tr><td></td><td colspan="3">Example 2:
– RESOURCE STATION_2 as configured in figure 20
– Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (note 4)
– INT_2 is triggered at t = 25, 50, 90, ... ms
– STATION_2 starts at t = 0</td></tr>
<tr><td></td><td colspan="3">Schedule</td></tr>
<tr><td></td><td>t(ms)</td><td>Executing</td><td>Waiting</td></tr>
<tr><td></td><td>0</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>25</td><td>P1 @ 2</td><td>P4.FB1 @ 2, P4 @ 1</td></tr>
<tr><td></td><td>30</td><td>P4 @ 1</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>35</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td></td><td>50</td><td>P4 @ 1</td><td>P1 @ 2, P4.FB1 @ 2</td></tr>
<tr><td></td><td>55</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td></td><td>85</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td></td><td>90</td><td>P4.FB1 @ 2</td><td>P4 @ 1</td></tr>
<tr><td></td><td>95</td><td>P4 @ 1</td><td></td></tr>
<tr><td></td><td>100</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>5b</td><td colspan="3">Preemptive scheduling</td></tr>
<tr><td></td><td colspan="3">Example 3:
– RESOURCE STATION_1 as configured in figure 20
– Execution times: P1 = 2 ms, P2 = 8 ms; P2.FB1 = P2.FB2 = 2 ms (note 3)
– STATION_1 starts at t = 0</td></tr>
<tr><td></td><td colspan="3">Schedule</td></tr>
<tr><td></td><td>t(ms)</td><td>Executing</td><td>Waiting</td></tr>
<tr><td></td><td>0</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
<tr><td></td><td>2</td><td>P1 @ 2</td><td>P2.FB1 @ 2, P2</td></tr>
<tr><td></td><td>4</td><td>P2.FB1 @ 2</td><td>P2</td></tr>
<tr><td></td><td>6</td><td>P2</td><td></td></tr>
<tr><td></td><td>10</td><td>P2.FB2 @ 1</td><td>P2</td></tr>
<tr><td></td><td>12</td><td>P2</td><td></td></tr>
<tr><td></td><td>16</td><td>P2</td><td>(P2 restarts)</td></tr>
<tr><td></td><td>20</td><td>P2.FB2 @ 1</td><td>P1 @ 2, P2.FB1 @ 2, P2</td></tr>
</table>

(continued on page 219)

Tableau 50 *(fin)*

<table>
<tr><th>N°</th><th colspan="3">Description/Exemples</th></tr>
<tr><td rowspan="12">5b</td><td colspan="3">Exemple 4:
– RESOURCE STATION_2 telle que configurée à la figure 20
– Temps d'exécution: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (note 4)
– INT_2 est déclenchée à t = 25, 50, 90, ... ms
– STATION_2 démarre à t = 0</td></tr>
<tr><td colspan="3">Calendrier</td></tr>
<tr><td>t(ms)</td><td>En cours d'exécution</td><td>En attente</td></tr>
<tr><td>0</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>25</td><td>P4 @ 1</td><td>P1. @ 2, P4.FB1 @ 2</td></tr>
<tr><td>30</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>35</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>50</td><td>P4 @ 1</td><td>P1 @ 2, P4.FB1 @ 2</td></tr>
<tr><td>55</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>85</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>90</td><td>P4. @ 1</td><td>P4 FB1 @ 2</td></tr>
<tr><td>95</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>100</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td colspan="4">NOTES
1 Les détails relatifs aux déclarations RESOURCE et PROGRAM ne sont pas indiqués; se reporter à 2.7. et 2.7.1.
2 La notation "X @ Y" indique que l'unité d'organisation de programme X est programmée ou qu'elle est exécutée au niveau de priorité Y.
3 Les temps d'exécution de P2.FB1 et de P2.FB2 ne sont pas inclus dans le temps d'exécution de P2.
4 Le temps d'exécution de P4.FB1 n'est pas inclus dans le temps d'exécution de P4.</td></tr>
</table>

Table 50 *(concluded)*

<table>
<tr><th>No.</th><th colspan="3">Description/Examples</th></tr>
<tr><td rowspan="12">5b</td><td colspan="3">Example 4:
– RESOURCE STATION_2 as configured in figure 20
– Execution times: P1 = 30 ms, P4 = 5 ms, P4.FB1 = 10 ms (note 4)
– INT_2 is triggered at = 25, 50, 90, ... ms
– STATION_2 starts at t = 0</td></tr>
<tr><td colspan="3">Schedule</td></tr>
<tr><td>t(ms)</td><td>Executing</td><td>Waiting</td></tr>
<tr><td>0</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>25</td><td>P4 @ 1</td><td>P1. @ 2, P4.FB1 @ 2</td></tr>
<tr><td>30</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>35</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>50</td><td>P4 @ 1</td><td>P1 @ 2, P4.FB1 @ 2</td></tr>
<tr><td>55</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td>85</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>90</td><td>P4. @ 1</td><td>P4 FB1 @ 2</td></tr>
<tr><td>95</td><td>P4.FB1 @ 2</td><td></td></tr>
<tr><td>100</td><td>P1 @ 2</td><td>P4.FB1 @ 2</td></tr>
<tr><td colspan="4">NOTES
1 Details of RESOURCE and PROGRAM declarations are not shown; see 2.7 and 2.7.1.
2 The notation "X @ Y" indicates that program organization unit X is scheduled or executing at priority Y.
3 The execution times of P2.FB1 and P2.FB2 are not included in the execution time of P2.
4 The execution time of P4.FB1 is not included in the execution time of P4.</td></tr>
</table>

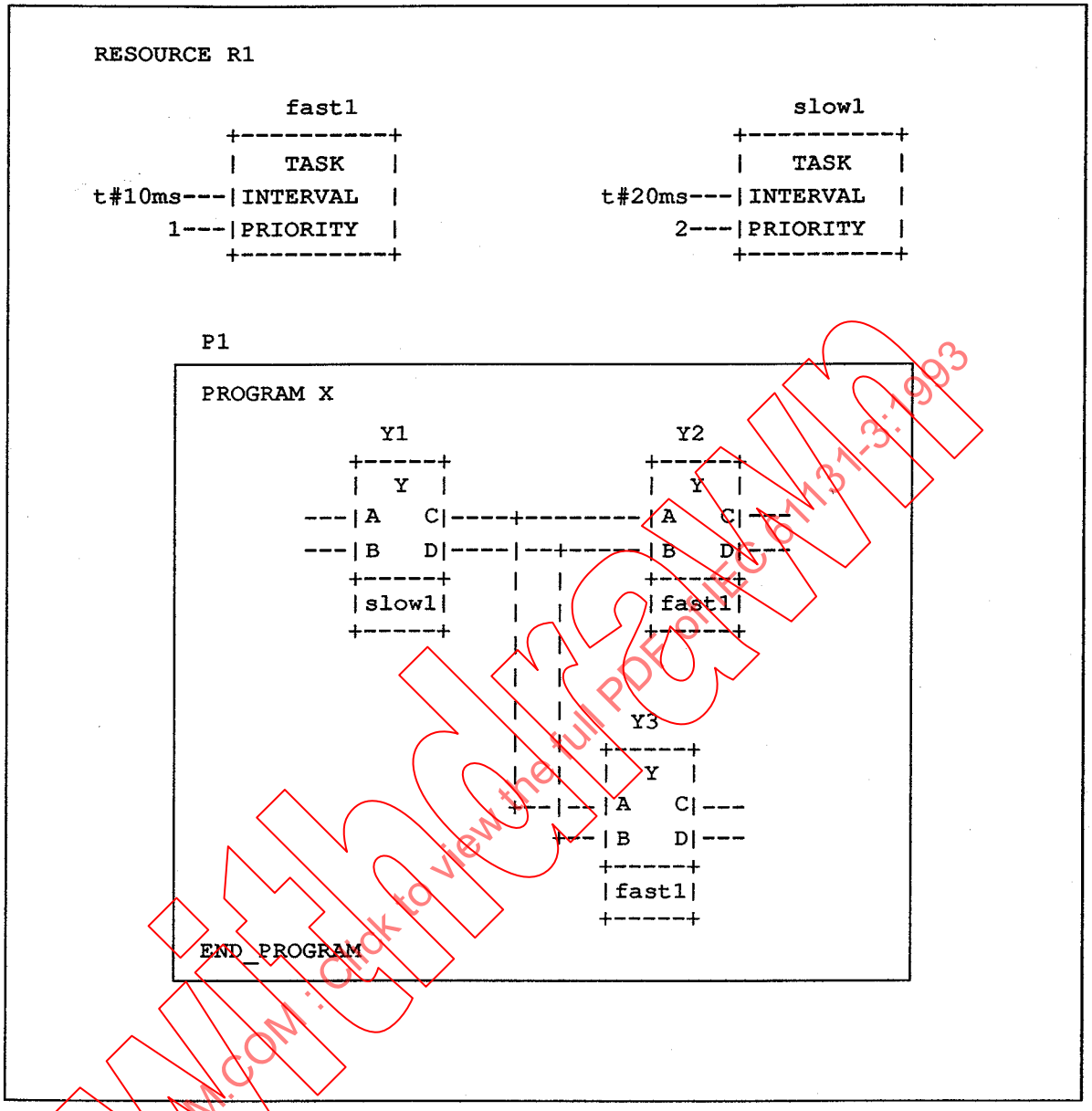


Figure 21a – Synchronisation de blocs fonctionnels avec associations de tâches explicites

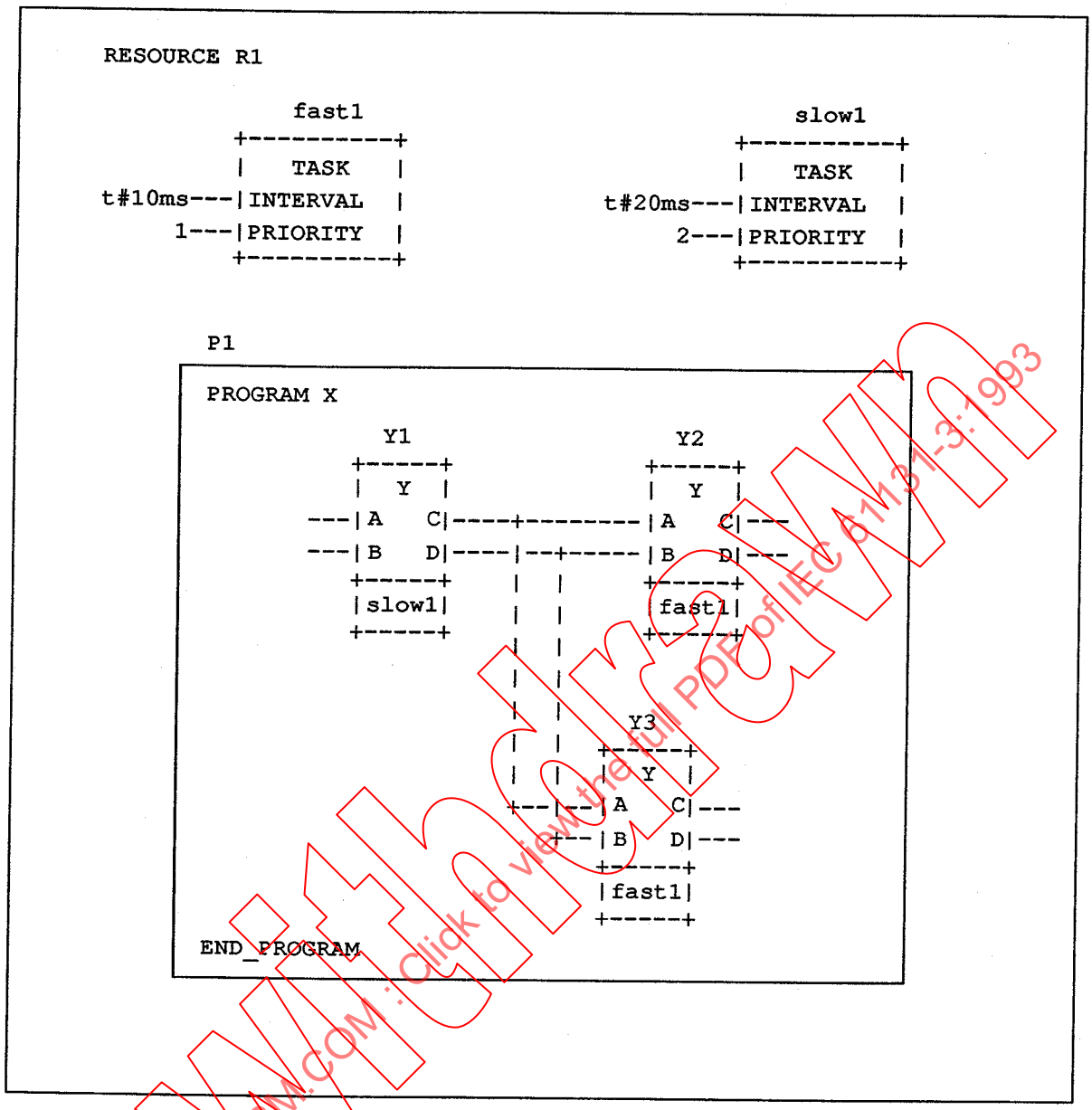


Figure 21a – Synchronization of function blocks with explicit task associations

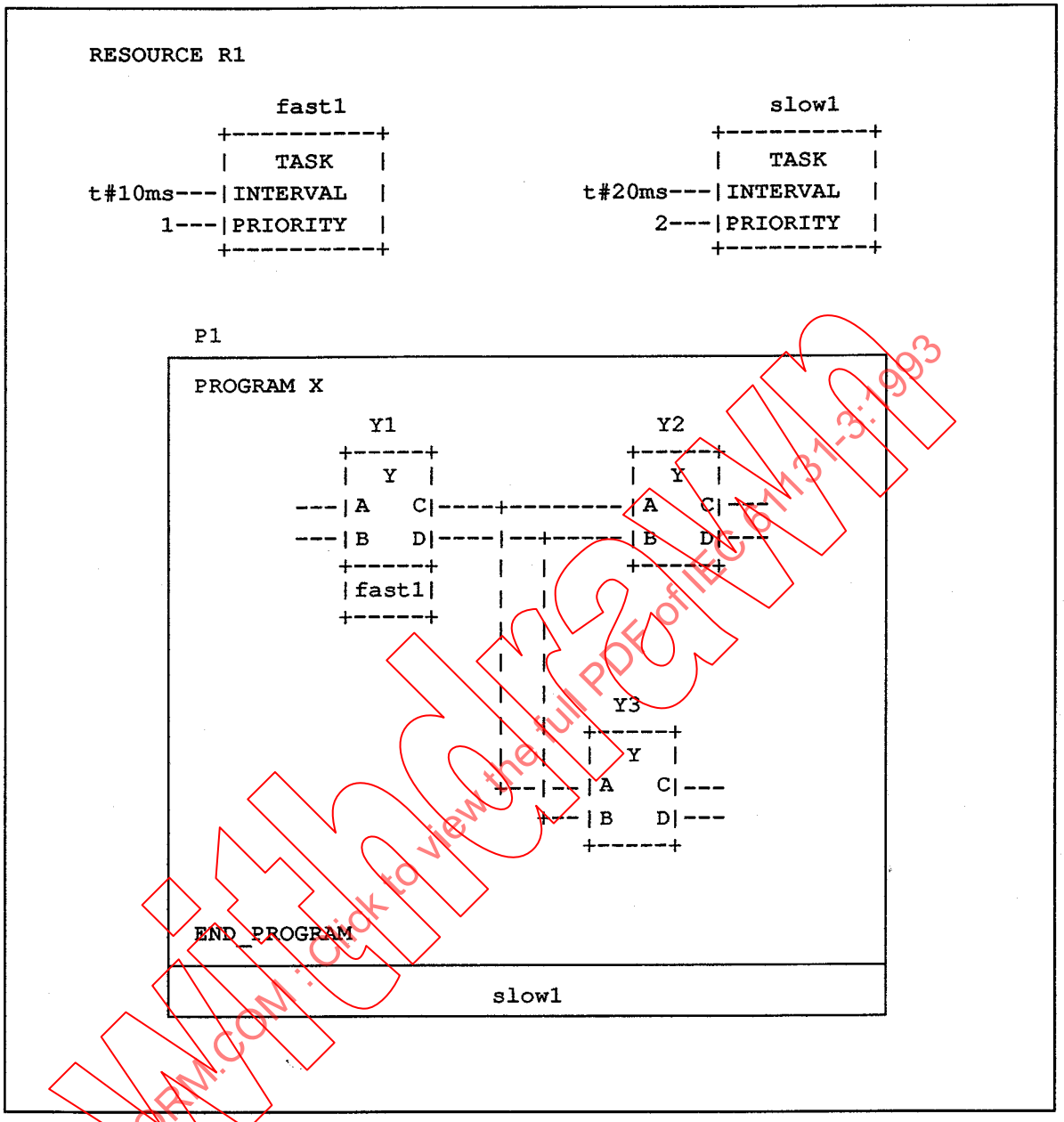


Figure 21b – Synchronisation de blocs fonctionnels avec associations de tâches implicites

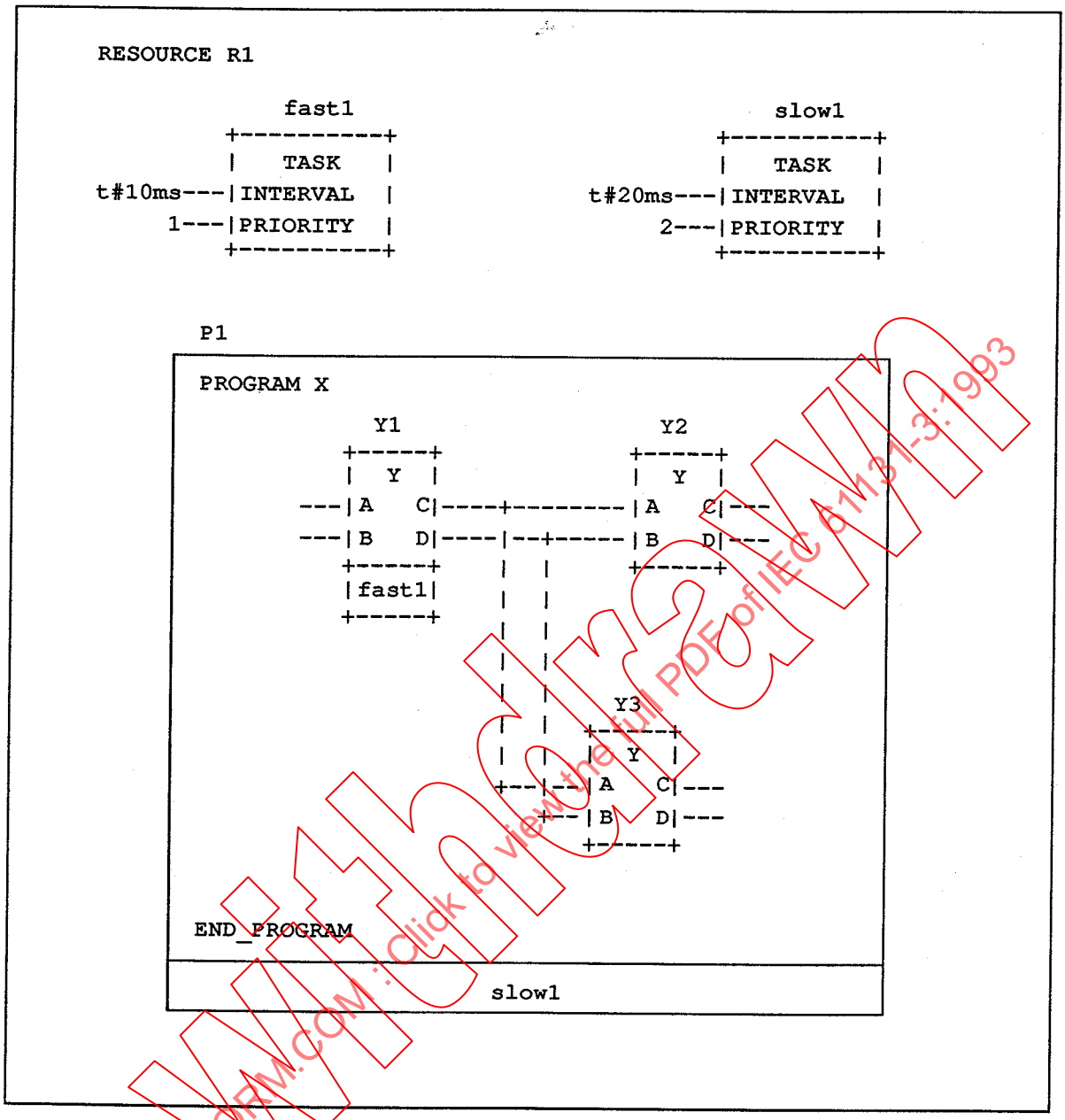


Figure 21b – Synchronization of function blocks with implicit task associations

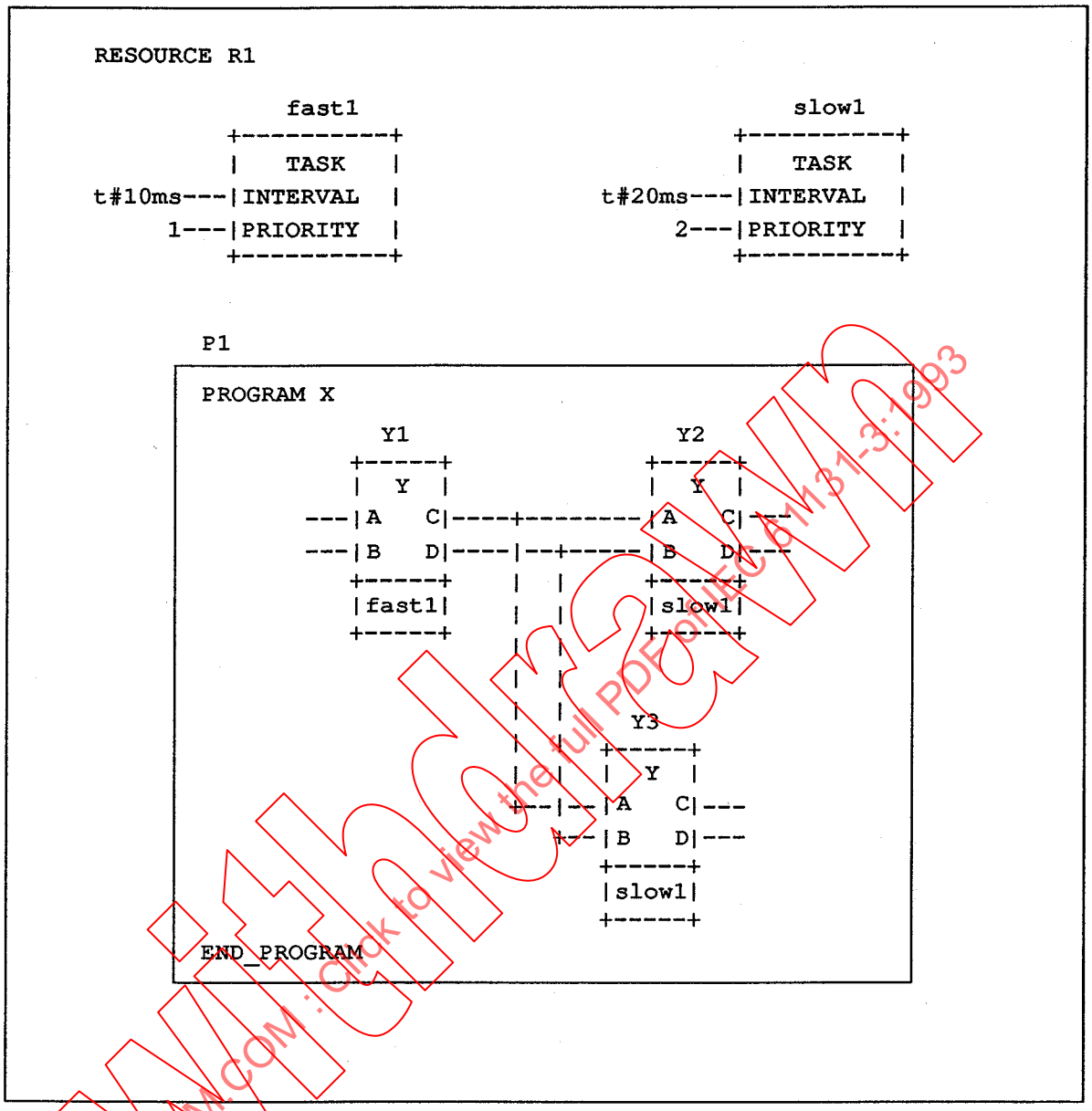


Figure 21c – Association de tâches explicites équivalentes à la figure 21b

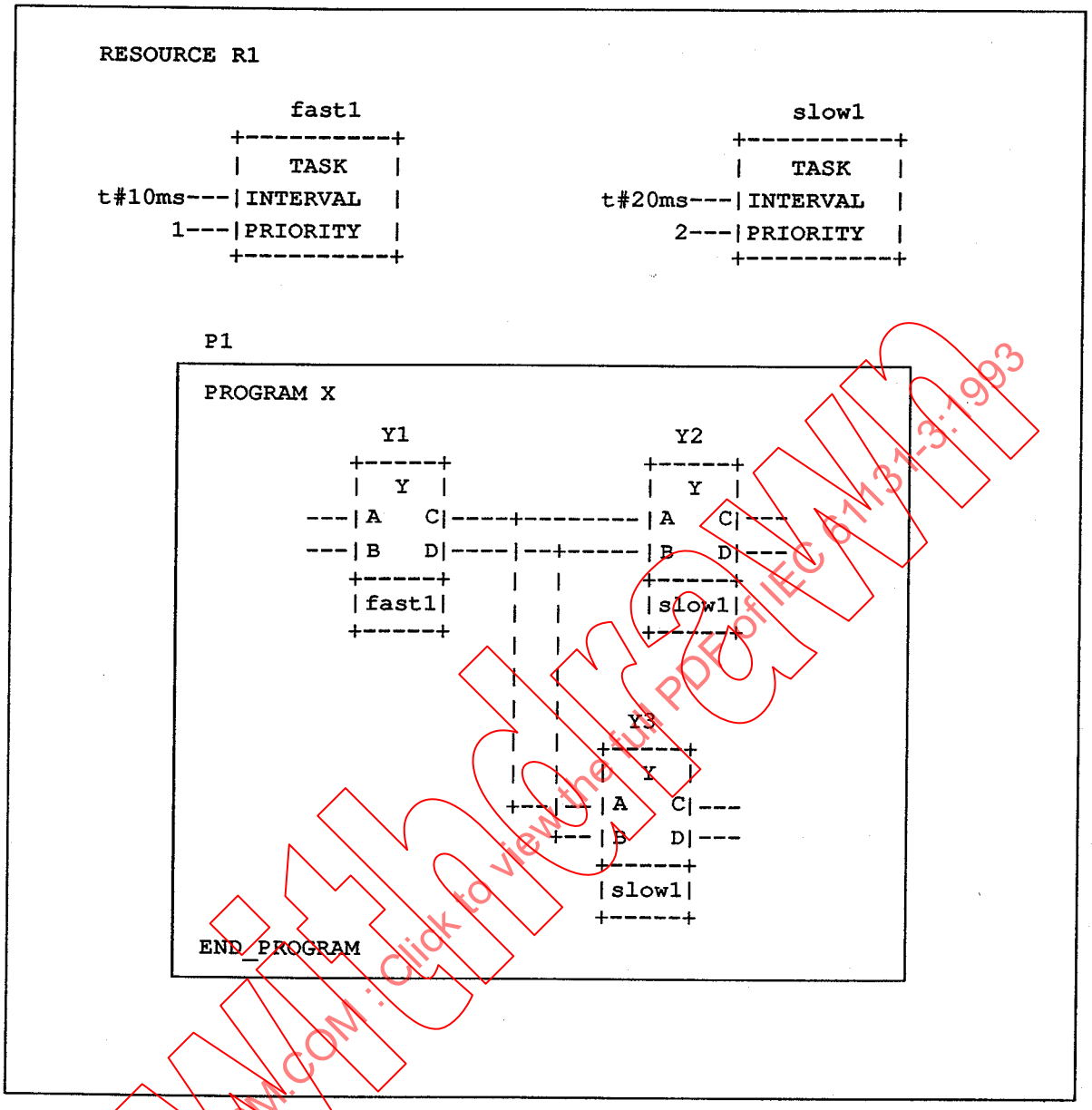


Figure 21c – Explicit task associations equivalent to figure 21b

3 Langages littéraux

Les langages littéraux définis dans cette norme sont le langage IL (liste d'instructions) et le langage ST littéral structuré). Les éléments du schéma de diagramme fonctionnel en séquence (SFC) définis en 2.6 peuvent être utilisés conjointement à l'un de ces deux langages.

3.1 *Eléments Communs*

Les éléments littéraux définis à l'article 2 doivent être communs aux langages littéraux (IL et ST) définis dans le présent article. En particulier, les éléments de structuration des programmes suivants doivent être communs aux langages littéraux:

TYPE...END_TYPE	(2.3.3)
VAR...END_VAR	(2.4.3)
VAR_INPUT...END_VAR	(2.4.3)
VAR_OUTPUT...END_VAR	(2.4.3)
VAR_IN_OUT...END_VAR	(2.4.3)
VAR_EXTERNAL...END_VAR	(2.4.3)
FUNCTION...END_FUNCTION	(2.5.1.3)
FUNCTION_BLOCK...END_FUNCTION_BLOCK	(2.5.2.2)
PROGRAM...END_PROGRAM	(2.5.3)
STEP...END_STEP	(2.6.2)
TRANSITION...END_TRANSITION	(2.6.3)
ACTION...END_ACTION	(2.6.4)

3.2 *Langage IL (liste d'instructions)*

Le présent paragraphe définit la sémantique du langage IL (Liste d'Instructions) dont la syntaxe formelle est donnée dans l'annexe B.2.

3.2.1 *Instructions*

Comme l'indique le Tableau 51, une *liste d'instructions* est composée d'une suite d'*instructions.* Chaque instruction doit débuter sur une nouvelle ligne et doit contenir un *opérateur* accompagné de modificateurs optionnels et, si cela est nécessaire pour l'opération considérée, un ou plusieurs opérandes séparés par des virgules. Les opérandes peuvent être choisis parmi les représentations de données définies en 2.2 pour les libellés, et en 2.4 pour les variables.

L'instruction peut être précédée d'une *étiquette* d'identification suivie de deux points (:). Si un *commentaire,* tel que défini en 2.1.5 est présent, il doit constituer le dernier élément d'une ligne. Des lignes vides peuvent être insérées entre les instructions.

Tableau 51 – Exemples de champs d'instruction

Etiquette	Opérateur	Opérande	Commentaire
START:	LD	%IX1	(* BOUTON POUSSOIR *)
	ANDN	%MX5	(* NON INHIBÉE *)
	ST	%QX2	(* MARCHE VENTILATEUR *)

3 Textual languages

The textual languages defined in this standard are IL (Instruction List) and ST (Structured Text). The sequential function chart (SFC) elements defined in 2.6 can be used in conjunction with either of these languages.

3.1 *Common elements*

The textual elements specified in clause 2 shall be common to the textual languages (IL and ST) defined in this clause. In particular, the following program structuring elements shall be common to textual languages:

TYPE...END_TYPE	(2.3.3)
VAR...END_VAR	(2.4.3)
VAR_INPUT...END_VAR	(2.4.3)
VAR_OUTPUT...END_VAR	(2.4.3)
VAR_IN_OUT...END_VAR	(2.4.3)
VAR_EXTERNAL...END_VAR	(2.4.3)
FUNCTION...END_FUNCTION	(2.5.1.3)
FUNCTION_BLOCK...END_FUNCTION_BLOCK	(2.5.2.2)
PROGRAM...END_PROGRAM	(2.5.3)
STEP...END_STEP	(2.6.2)
TRANSITION...END_TRANSITION	(2.6.3)
ACTION...END_ACTION	(2.6.4)

3.2 *Language IL (Instruction List)*

This subclause defines the semantics of the IL (Instruction List) language whose formal syntax is given in B.2.

3.2.1 *Instructions*

As illustrated in table 51, an *instruction list* is composed of a sequence of *instructions*. Each instruction shall begin on a new line and shall contain an operator with optional *modifiers*, and, if necessary for the particular operation, one or more *operands* separated by commas. Operands can be any of the data representations defined in 2.2 for literals and 2.4 for variables.

The instruction can be preceded by an identifying *label* followed by a colon (:). A *comment*, as defined in 2.1.5, if present, shall be the last element on a line. Empty lines can be inserted between instructions.

Table 51 – Examples of instruction fields

Label	Operator	Operand	Comment
START:	LD	%IX1	(* PUSH BUTTON *)
	ANDN	%MX5	(* NOT INHIBITED*)
	ST	%QX2	(* FAN ON *)

3.2.2 *Opérateurs, modificateurs et opérandes*

Les opérateurs standards ainsi que leurs modificateurs et opérandes autorisés doivent être tels qu'énumérés dans le tableau 52. La saisie des opérateurs doit respecter les conventions exposées en 2.5.1.4.

Sauf mention contraire dans le tableau 52, la sémantique des opérateurs doit être la suivante:

résultat := résultat OP opérande

Cela signifie que la valeur de l'expression faisant l'objet de l'évaluation est remplacée par sa valeur actuelle sur laquelle intervient l'opérateur par rapport à l'opérande. Par exemple, l'instruction AND %IX1 est interprétée comme:

résultat := résultat AND %IX1

Les opérateurs de comparaison doivent être interprétés comme ayant le résultat courant à gauche de la comparaison et l'opérande à droite, le résultat obtenu étant booléen. Par exemple, l'instruction "GT %IW10" aura comme résultat la valeur booléenne 1, si le résultat courant est supérieur à la valeur du mot d'entrée 10, et la valeur booléenne 0 dans tous les autres cas.

Le modificateur "N" indique une négation booléenne de l'opérande. Par exemple, l'instruction ANDN %IX2 est interprétée de la manière suivante:

résultat := résultat AND NOT %IX2

Le modificateur parenthèse gauche "(" indique que l'évaluation de l'opérateur doit être différée jusqu'à ce qu'un opérateur parenthèse droit ")" soit rencontré; par exemple, la séquence d'instructions suivante:

```
AND(  %IX1
OR    %IX2
)
```

doit être interprétée de la manière suivante:

résultat := résultat AND (%IX1 OR %IX2)

Le modificateur "C" indique que l'instruction donnée ne doit être exécutée que si le résultat faisant l'objet de l'évaluation en cours a la valeur booléenne 1 (ou la valeur booléenne 0 si l'opérateur est combiné avec le modificateur "N").

3.2.2 *Operators, modifiers and operands*

Standard operators with their allowed modifiers and operands shall be as listed in table 52. The typing of operators shall conform to the conventions of 2.5.1.4.

Unless otherwise defined in table 52, the semantics of the operators shall be

result := result OP operand

That is, the value of the expression being evaluated is replaced by its current value operated upon by the operator with respect to the operand. For instance, the instruction AND %IX1 is interpreted as

result := result AND %IX1

The comparison operators shall be interpreted with the current result to the left of the comparison and the operand to the right, with a Boolean result. For instance, the instruction "GT %IW10" will have the Boolean result 1 if the current result is greater than the value of Input Word 10, and the Boolean result 0 otherwise.

The modifier "N" indicates Boolean negation of the operand. For instance, the instruction ANDN %IX2 is interpreted as

result := result AND NOT %IX2

The left parenthesis modifier "(" indicates that evaluation of the operator shall be deferred until a right parenthesis operator ")" is encountered, e.g., the sequence of instructions

```
AND(  %IX1
OR    %IX2
)
```

shall be interpreted as

result := result AND (%IX1 OR %IX2)

The modifier "C" indicates that the associated instruction shall be performed only if the value of the currently evaluated result is Boolean 1 (or Boolean 0 if the operator is combined with the "N" modifier)

Tableau 52 – Opérateurs de liste d'instructions

N°	Opérateur	Modificateurs	Opérande	Sémantique
1	LD	N	Note 2	Rendre le résultat courant égal à l'opérande
2	ST	N	Note 2	Mémoriser le résultat à l'emplacement de l'opérande
3	S R	Note 3 Note 3	BOOL BOOL	Positionner l'opérande booléen à 1 Remettre l'opérande booléen à 0
4 5 6 7	AND & OR XOR	N, (N, (N, (N, (	BOOL BOOL BOOL BOOL	AND booléen AND booléen OR booléen OR exclusif booléen
8 9 10 11	ADD SUB MUL DIV	((((	Note 2 Note 2 Note 2 Note 2	Addition Soustraction Multiplication Division
12 13 14 15 16 17	GT GE EQ NE LE LT	((((((	Note 2 Note 2 Note 2 Note 2 Note 2 Note 2	Comparaison: > Comparaison: >= Comparaison: = Comparaison: <> Comparaison: <= Comparaison: <
18 19 20	JMP CAL RET	C, N C, N C, N	LABEL NAME	Saut vers l'étiquette Appel d'un bloc fonctionnel (note 4) Retour d'une fonction appelée ou d'un bloc fonctionnel
21	)			Evaluation d'une opération différée

NOTES

1 Se reporter au texte précédent pour toute explication relative aux modificateurs et à l'évaluation des expressions.

2 Ces opérateurs doivent être soit surchargés soit saisis comme défini en 2.5.1.4. Le résultat courant et l'opérande doivent être du même type.

3 Ces opérations sont effectuées si et seulement si le résultat courant a la valeur booléenne 1.

4 Le nom du bloc fonctionnel est suivi par une liste d'arguments entre parenthèses, telle que définie en 3.2.3.

5 Lorsqu'une instruction JMP est contenue dans une construction ACTION...END_ACTION, l'opérande doit être une étiquette à l'intérieur de la même construction.

3.2.3 *Fonctions et blocs fonctionnels*

Les fonctions, telles que définies en 2.5.1 doivent être lancées en inscrivant le nom de la fonction dans le champ opérateur. Le résultat courant doit être utilisé comme premier argument de la fonction. Si nécessaire, des arguments supplémentaires doivent être fournis dans le champ opérande. La valeur renvoyée par une fonction, après exécution correcte d'une instruction RET ou lorsque la fin physique de la fonction est atteinte, doit devenir le "résultat courant" tel qu'il est décrit en 3.2.2.

Les blocs fonctionnels, tels que définis en 2.5.2, peuvent être lancés sous condition ou inconditionnellement à l'aide de l'opérateur CAL (Appel) figurant dans le tableau 52. Comme l'illustre le tableau 53, ce lancement peut prendre trois formes. Les opérateurs d'entrée indiqués dans le tableau 54 peuvent être utilisés conjointement à la caractéristique 3 du tableau 53.

Table 52 – Instruction List (IL) operators

No.	Operator	Modifiers	Operand	Semantics
1	LD	N	Note 2	Set current result equal to operand
2	ST	N	Note 2	Store current result to operand location
3	S R	Note 3 Note 3	BOOL BOOL	Set Boolean operand to 1 Reset Boolean operand to 0
4 5 6 7	AND & OR XOR	N, (N, (N, (N, (	BOOL BOOL BOOL BOOL	Boolean AND Boolean AND Boolean OR Boolean Exclusive OR
8 9 10 11	ADD SUB MUL DIV	((((	Note 2 Note 2 Note 2 Note 2	Addition Subtraction Multiplication Division
12 13 14 15 16 17	GT GE EQ NE LE LT	((((((	Note 2 Note 2 Note 2 Note 2 Note 2 Note 2	Comparison: >; Comparison: >= Comparison: = Comparison: <>; Comparison: <= Comparison: <
18 19 20	JMP CAL RET	C, N C, N C, N	LABEL NAME	Jump to label Call function block (note 4) Return from called function or function block
21	)			Evaluate deferred operation

NOTES

1 See 3.2.2 for explanation of modifiers and evaluation of expressions.

2 These operators shall be either overloaded or typed as defined in 2.5.1.4. The current result and the operand shall be of the same type.

3 These operations are performed if and only if the value of the current result is Boolean 1.

4 The function block name is followed by a parenthesized argument list as defined in 3.2.3.

5 When a JMP instruction is contained in an ACTION... END_ACTION construct, the operand shall be a label within the same construct.

3.2.3 *Functions and function blocks*

Functions as defined in 2.5.1 shall be invoked by placing the function name in the operator field. The current result shall be used as the first argument of the function. Additional arguments, if required, shall be given in the operand field. The value returned by a function upon the successful execution of a RET instruction or upon reaching the physical end of the function shall become the "current result" described in 3.2.2.

Function blocks as defined in 2.5.2 can be invoked conditionally and unconditionally via the CAL (Call) operator listed in table 52. As shown in table 53, this invocation can take one of three forms. The input operators shown in table 54 can be used in conjunction with feature 3 of table 53.

Tableau 53 – Caractéristiques du lancement de bloc fonctionnel en langage IL

N°	Description/exemple
1	CAL avec liste d'entrées: `CAL C10(CU := %IX10, PV := 15)`

2	CAL avec entrées de charge/mémoire
	`LD 15`
	`ST C10.PV`
	`LD %IX10`
	`ST C10.CU`
	`CAL C10`

3	Utilisation d'opérateurs d'entrée:
	`LD 15`
	`PV C10`
	`LD %IX10`
	`CU C10`

NOTE - Une déclaration telle que VAR C10 : CTU ; END_VAR est supposée dans les exemples ci-dessus.	

Tableau 54 – Opérateurs d'entrée standards des blocs fonctionnels en langage IL

N°	Opérateurs	Type FB	Référence
4	S1,R	SR	2.5.2.3.1
5	S,R1	RS	2.5.2.3.1
6	CLK	R_TRIG	2.5.2.3.2
7	CLK	F_TRIG	2.5.2.3.2
8	CU,R,PV	CTU	2.5.2.3.3
9	CD,LD,PV	CTD	2.5.2.3.3
10	CU,CD,R,LD,PV	CTUD	2.5.2.3.3
11	IN,PT	TP	2.5.2.3.4
12	IN,PT	TON	2.5.2.3.4
13	IN,PT	TOF	2.5.2.3.4

3.3 *Langage ST (littéral structuré)*

Le présent paragraphe définit la sémantique du langage ST (littéral structuré), dont la syntaxe est définie dans l'annexe B.3. Dans ce langage, la fin d'une ligne littérale doit être traitée de la même manière qu'un caractère d'espacement (SP), tel que défini en 2.1.4.

3.3.1 *Expressions*

Une *expression* est une construction syntaxique qui, lorsqu'elle est évaluée, fournit une valeur correspondant à l'un des types de données définis en 2.3.1 et 2.3.3.

Table 53 – Function block invocation features for IL language

No.	Description/example
1	CAL with input list: `CAL C10(CU := %IX10, PV := 15)`

2	CAL with load/store of inputs:
	`LD 15`
	`ST C10.PV`
	`LD %IX10`
	`ST C10.CU`
	`CAL C10`

3	Use of input operators:
	`LD 15`
	`PV C10`
	`LD %IX10`
	`CU C10`

NOTE - A declaration such as VAR C10 : CTU ; END_VAR is assumed in the above examples.	

Table 54 – Standard function block input operators for IL language

No.	Operators	FB Type	Reference
4	S1,R	SR	2.5.2.3.1
5	S,R1	RS	2.5.2.3.1
6	CLK	R_TRIG	2.5.2.3.2
7	CLK	F_TRIG	2.5.2.3.2
8	CU,R,PV	CTU	2.5.2.3.3
9	CD,LD,PV	CTD	2.5.2.3.3
10	CU,CD,R,LD,PV	CTUD	2.5.2.3.3
11	IN,PT	TP	2.5.2.3.4
12	IN,PT	TON	2.5.2.3.4
13	IN,PT	TOF	2.5.2.3.4

3.3 *Language ST (Structured Text)*

This subclause defines the semantics of the ST (Structured Text) language whose syntax is defined in B.3. In this language, the end of a textual line shall be treated the same as a space (SP) character, as defined in 2.1.4.

3.3.1 *Expressions*

An *expression* is a construct which, when evaluated, yields a value corresponding to one of the data types defined in 2.3.1 and 2.3.3.

Les expressions sont composées d'opérateurs et d'opérandes. Un *opérande* doit être un libellé conforme à la définition donnée en 2.2, une variable telle que définie en 2.4 ou une autre expression.

Les *opérateurs* du langage ST sont résumés dans le tableau 55. L'évaluation d'une expression consiste à appliquer les opérateurs aux opérandes dans l'ordre défini par l'opérateur *priorité* du tableau 55. L'opérateur ayant la priorité la plus élevée dans une expression doit être appliqué en premier; il sera suivi par l'opérateur ayant la priorité immédiatement inférieure et ainsi de suite, jusqu'à la fin de l'évaluation. Les opérateurs bénéficiant d'une même priorité doivent être appliqués dans l'ordre où ils apparaissent dans l'expression, de gauche à droite. Par exemple, si A, B, C et D sont du type INT et ont respectivement les valeurs 1, 2, 3 et 4, alors

A+B–C*ABS(D)

doit prendre la valeur –9, et

(A+B–C)*ABS(D)

doit prendre la valeur 0.

Lorsqu'un opérateur a deux opérandes, l'opérande situé le plus à gauche doit être évalué le premier. Par exemple, dans l'expression

SIN(A)*COS(B)

l'expression SIN(A) doit être évaluée la première, suivie par COS(B), puis par l'évaluation du produit.

Les expressions booléennes ne peuvent être évaluées que dans la mesure où cela est nécessaire pour déterminer la valeur résultante. Par exemple, si A<=B, seule l'expression (A>B) devrait être évaluée pour déterminer que la valeur de l'expression

(A>B) & (C<D)

est un zéro booléen.

Les fonctions doivent être lancées sous forme d'éléments d'expressions composés du nom de la fonction suivi d'une liste d'arguments entre parenthèses, comme défini en 2.5.1.1.

Lorsqu'un opérateur dans une expression peut être représenté par l'une des fonctions surchargées, telles que définies en 2.5.1.5, la conversion des opérandes et des résultats doit se conformer à la règle et aux exemples donnés en 2.5.1.4.

Expressions are composed of operators and operands. An *operand* shall be a literal as defined in 2.2, a variable as defined in 2.4, a function invocation as defined in 2.5.1, or another expression.

The *operators* of the ST language are summarized in table 55. The evaluation of an expression consists of applying the operators to the operands in a sequence defined by the operator precedence shown in table 55. The operator with highest precedence in an expression shall be applied first, followed by the operator of next lower precedence, etc., until evaluation is complete. Operators of equal precedence shall be applied as written in the expression from left to right. For example, if A, B, C, and D are of type INT with values 1, 2, 3, and 4, respectively, then

```
A+B–C*ABS(D)
```

shall evaluate to –9, and

```
(A+B–C)*ABS(D)
```

shall evaluate to 0 .

When an operator has two operands, the leftmost operand shall be evaluated first. For example, in the expression

```
SIN(A)*COS(B)
```

the expression SIN(A) shall be evaluated first, followed by COS(B), followed by evaluation of the product.

Boolean expressions may be evaluated only to the extent necessary to determine the resultant value. For instance, if A<=B, then only the expression (A>B) would be evaluated to determine that the value of the expression

```
(A>B) & (C<D)
```

is Boolean zero.

Functions shall be invoked as elements of expressions consisting of the function name followed by a parenthesized list of arguments, as defined in 2.5.1.1.

When an operator in an expression can be represented as one of the overloaded functions defined in 2.5.1.5, conversion of operands and results shall follow the rule and examples given in 2.5.1.4.

Tableau 55 – Opérateurs du langage ST

N°	Opération	Symbole	Priorité
1	Mise entre parenthèses	(Expression)	MAXIMALE
2	Evaluation de Fonction Exemples:	Identificateur (liste d'arguments) LN(A), MAX(X,Y), etc.	
3	Exponentiation	**	
4 5	Négation Complément	– NOT	
6 7 8	Multiplication Division Modulo	* / MOD	
9 10	Addition Soustraction	+ –	
11	Comparaison	< , > , <= , >=	
12 13	Egalité Inégalité	= <>	
14 15	AND booléen AND booléen	& AND	
16	OR exclusif booléen	XOR	
17	OR booléen	OR	MINIMALE

NOTES

1 Les mêmes restrictions s'appliquent aux opérandes de ces opérateurs et aux entrées des fonctions correspondantes définies en 2.5.1.5.

2 Le résultat de l'évaluation de A**B doit être le même que le résultat de l'évaluation de la fonction EXPT (A, B) telle que définie au tableau 24.

Table 55 – Operators of the ST language

No.	Operation	Symbol	Precedence
1	Parenthesization	(Expression)	HIGHEST
2	Function evaluation Examples:	Identifier (argument list) LN(A), MAX(X,Y), etc.	
3	Exponentiation	**	
4 5	Negation Complement	– NOT	
6 7 8	Multiply Divide Modulo	* / MOD	
9 10	Add Subtract	+ –	
11	Comparison	< , > , <= , >=	
12 13	Equality Inequality	= <>	
14 15	Boolean AND Boolean AND	& AND	
16	Boolean Exclusive OR	XOR	
17	Boolean OR	OR	LOWEST

NOTES

1 The same restrictions apply to the operands of these operators as to the inputs of the corresponding functions defined in 2.5.1.5.

2 The result of evaluating the expression A**B shall be the same as the result of evaluating the function EXPT(A, B) as defined in table 24.

3.3.2 *Enoncés*

Les énoncés du langage ST sont résumés dans le tableau 56. Les énoncés doivent se terminer par des points virgules, comme spécifié dans la syntaxe de l'annexe B.3.

Tableau 56 – Enoncés du langage ST

N°	Type d'énoncé/référence	Exemples
1	Affectation (3.3.2.1)	A := B ; CV := CV+1 ; C := SIN(X) ;
2	Lancement d'un bloc fonctionnel et utilisation de sortie FB (3.3.2.2)	CMD_TMR(IN := %IX5, PT := T#300ms) ; A := CMD_TMR.Q ;
3	RETURN (3.3.2.2)	RETURN ;
4	IF (3.3.2.3)	D := B*B - 4*A*C ; IF D < 0.0 THEN NROOTS := 0 ; ELSIF D = 0.0 THEN NROOTS := 1 ; X1 := - B/ (2.0*A) ; ELSE NROOTS := 2 ; X1 := (-B+SQRT(D))/(2.0*A) ; X2 := (-B-SQRT(D))/(2.0*A) ; END_IF ;
5	CASE (3.3.2.3)	TW := BCD_TO_INT(THUMBWHEEL) ; TW_ERROR := 0 ; CASE TW OF 1,5 : DISPLAY := OVEN_TEMP ; 2 : DISPLAY := MOTOR_SPEED ; 3 : DISPLAY := GROSS_TARE ; 4,6..10 : DISPLAY := STATUS (TW-4) ; ELSE DISPLAY := 0 ; TW_ERROR := 1 ; END_CASE ; QW100 := INT_TO_BCD(DISPLAY) ;

6	FOR (3.3.2.4)	J := 101 ;
		FOR I := 1 TO 100 BY 2 DO
		IF WORDS[I] = 'KEY' THEN
		J := I ;
		EXIT ;
		END_IF ;
		END_FOR ;

7	WHILE (3.3.2.4)	J := 1 ; WHILE J <= 100 & WORDS[J] <> 'KEY' DO J := J+2 ; END_WHILE ;
8	REPEAT (3.3.2.4)	J := -1 ; REPEAT J := J+2 ; UNTIL J = 101 OR WORDS[J] = 'KEY' END_REPEAT ;
9	EXIT (3.3.2.4)	EXIT ;
10	Enoncé Vide	;

NOTE – Si l'énoncé EXIT (n° 9) est accepté, il doit alors être accepté pour tous les énoncés d'itération (FOR, WHILE, REPEAT) qui sont acceptés dans l'application concernée.

3.3.2 *Statements*

The statements of the ST language are summarized in table 56. Statements shall be terminated by semicolons as specified in the syntax of B.3.

Table 56 – ST language statements

No.	Statement type/Reference	Examples
1	Assignment (3.3.2.1)	A := B ; CV := CV+1 ; C := SIN(X) ;
2	Function block invocation and FB output usage (3.3.2.2)	CMD_TMR(IN := %IX5, PT := T#300ms) ; A := CMD_TMR.Q ;
3	RETURN (3.3.2.2)	RETURN ;

4	IF (3.3.2.3)	D := B*B - 4*A*C ;
		IF D < 0.0 THEN NROOTS := 0 ;
		ELSIF D = 0.0 THEN
		  NROOTS := 1 ;
		  X1 := - B/ (2.0*A) ;
		ELSE
		  NROOTS := 2 ;
		  X1 := (-B+SQRT(D))/(2.0*A) ;
		  X2 := (-B-SQRT(D))/(2.0*A) ;
		END_IF ;

5	CASE (3.3.2.3)	TW := BCD_TO_INT(THUMBWHEEL) ;
		TW_ERROR := 0 ;
		CASE TW OF
		  1,5 : DISPLAY := OVEN_TEMP ;
		  2 : DISPLAY := MOTOR_SPEED ;
		  3 : DISPLAY := GROSS_TARE ;
		  4,6..10 : DISPLAY := STATUS (TW-4) ;
		ELSE DISPLAY := 0 ;
		  TW_ERROR := 1 ;
		END_CASE ;
		QW100 := INT_TO_BCD(DISPLAY) ;

6	FOR (3.3.2.4)	J := 101 ;
		FOR I := 1 TO 100 BY 2 DO
		  IF WORDS[I] = 'KEY' THEN
		  J := I ;
		  EXIT ;
		  END_IF ;
		END_FOR ;

7	WHILE (3.3.2.4)	J := 1 ; WHILE J <= 100 & WORDS[J] <> 'KEY' DO J := J+2 ; END_WHILE ;
8	REPEAT (3.3.2.4)	J := -1 ; REPEAT J := J+2 ; UNTIL J = 101 OR WORDS[J] = 'KEY' END_REPEAT ;
9	EXIT (3.3.2.4)	EXIT ;
10	Empty Statement	;

NOTE - If the EXIT statement (9) is supported, then it shall be supported for all of the iteration statements (FOR, WHILE, REPEAT) which are supported in the implementation.

3.3.2.1 *Enoncés d'affectation*

L'énoncé d'affectation remplace la valeur actuelle d'une variable mono- ou multi-élément par le résultat de l'évaluation d'une expression. Un énoncé d'affectation doit se composer d'une référence de variable à gauche, suivie de l'*opérateur d'affectation* ":=", suivi de l'expression à évaluer. Par exemple, l'énoncé:

A := B ;

serait utilisé pour remplacer la valeur mono-élément de la variable A par la valeur actuelle de la variable B, sous réserve que ces deux valeurs soient de type INT. Cependant, si A et B sont toutes les deux du type ANALOG_CHANNEL_CONFIGURATION, comme décrit dans le tableau 12, les valeurs de tous les éléments de la variable structurée A seront alors remplacées par les valeurs actuelles des éléments correspondants de la variable B.

Comme indiqué dans la figure 6, l'énoncé d'affectation doit être également utilisé pour affecter la valeur à renvoyer par une fonction, en inscrivant le nom de la fonction à gauche d'un opérateur d'affectation dans le corps de la déclaration de fonction. La valeur renvoyée par la fonction doit être le résultat de l'évaluation la plus récente d'une telle affectation. C'est une erreur de revenir de l'évaluation d'une fonction dont la sortie "ENO" n'est pas nulle, sauf dans le cas où au moins une affectation a été effectuée.

3.3.2.2 *Enoncés de commande de fonctions et de blocs fonctionnels*

Les énoncés de commande de fonctions et de blocs fonctionnels intègrent les mécanismes nécessaires au lancement des blocs fonctionnels et à la reprise en main par l'entité appelante avant la fin physique d'une fonction ou d'un bloc fonctionnel.

L'évaluation d'une fonction doit être lancée comme partie de l'évaluation d'une expression, telle que spécifiée en 3.3.1.

Les blocs fonctionnels doivent être lancés par un énoncé comportant le nom du bloc fonctionnel suivi d'une liste, entre parenthèses, d'affectations nommées de valeurs de paramètres d'entrée, comme cela est illustré dans le tableau 56. L'ordre dans lequel les paramètres d'entrée sont énumérés dans le lancement d'un bloc fonctionnel ne doit pas être important. Il n'est pas impératif d'affecter des valeurs à tous les paramètres d'entrée à chaque lancement d'un bloc fonctionnel. Si, lors du lancement d'un bloc fonctionnel, un paramètre particulier ne se voit attribuer aucune valeur, la valeur préalablement affectée (ou la valeur initiale, si aucune affectation préalable n'a été effectuée) doit être appliquée.

L'énoncé RETURN doit permettre de sortir prématurément d'une fonction ou d'un bloc fonctionnel (par exemple, suite à l'évaluation d'un énoncé IF).

3.3.2.3 *Enoncés de sélection*

Les énoncés de sélection comprennent les énoncés IF et les énoncés CASE. Un énoncé de sélection choisit, en fonction d'une condition déterminée, l'un de ses énoncés constitutifs (ou un groupe d'entre eux) en vue de son exécution. Des exemples d'énoncés de sélection sont fournis dans le tableau 56.

3.3.2.1 *Assignment statements*

The assignment statement replaces the current value of a single or multi-element variable by the result of evaluating an expression. An assignment statement shall consist of a variable reference on the left-hand side, followed by the *assignment operator* ":=", followed by the expression to be evaluated. For instance, the statement

A := B ;

would be used to replace the single data value of variable A by the current value of variable B if both were of type INT. However, if both A and B were of type ANALOG_CHANNEL_CONFIGURATION as described in table 12, then the values of all the elements of the structured variable A would be replaced by the current values of the corresponding elements of variable B.

As illustrated in figure 6, the assignment statement shall also be used to assign the value to be returned by a function, by placing the function name to the left of an assignment operator in the body of the function declaration. The value returned by the function shall be the result of the most recent evaluation of such an assignment. It is an error to return from the evaluation of a function with the "ENO" output non-zero unless at least one such assignment has been made.

3.3.2.2 *Function and function block control statements*

Function and function block control statements consist of the mechanisms for invoking function blocks and for returning control to the invoking entity before the physical end of a function or function block.

Function evaluation shall be invoked as part of expression evaluation, as specified in 3.3.1.

Function blocks shall be invoked by a statement consisting of the name of the function block followed by a parenthesized list of named input parameter value assignments, as illustrated in table 56. The order in which input parameters are listed in a function block invocation shall not be significant. It is not required that all input parameters be assigned values in every invocation of a function block. If a particular parameter is not assigned a value in a function block invocation, the previously assigned value (or the initial value, if no previous assignment has been made) shall apply.

The RETURN statement shall provide early exit from a function or function block (e.g., as the result of the evaluation of an IF statement).

3.3.2.3 *Selection statements*

Selection statements include the IF and CASE statements. A selection statement selects one (or a group) of its component statements for execution, based on a specified condition. Examples of selection statements are given in table 56.

L'énoncé IF indique qu'un groupe d'énoncés ne doit être exécuté que si la valeur prise par l'expression booléenne associée est 1 (vraie). Si la condition n'est pas vérifiée, soit aucun énoncé n'est exécuté, soit le groupe d'énoncés suivi du mot clé ELSE (ou du mot clé ELSIF si la condition booléenne associée est vraie) doit être exécuté.

L'énoncé CASE se compose d'une expression qui doit être évaluée sur une variable de type INT (que l'on appelle le "sélecteur"), et d'une liste de groupes d'énoncés, chacun d'entre eux étant étiqueté par une ou plusieurs plages de valeurs entières. Cet énoncé précise que le premier groupe d'énoncés, dont l'une des plages contient la valeur calculée du sélecteur, doit être exécuté. Si la valeur du sélecteur n'apparaît dans aucune plage d'aucun cas, la suite d'énoncés qui suit le mot clé ELSE (s'il apparaît dans l'énoncé CASE) doit être exécuté. Dans le cas contraire, aucune des suites d'énoncés ne doit être exécutée.

3.3.2.4 *Enoncés d'itération*

Les énoncés d'itération indiquent que le groupe d'énoncés associés doit être exécuté de façon répétitive. L'énoncé FOR est utilisé si le nombre d'itérations peut être déterminé à l'avance; dans le cas contraire, les éléments WHILE ou REPEAT sont utilisés.

L'énoncé EXIT doit être utilisé pour mettre fin aux itérations avant que la condition terminale soit satisfaite.

Lorsque l'énoncé EXIT se trouve dans des constructions itératives imbriquées, la sortie doit être effectuée à partir de la boucle la plus proche dans laquelle se trouve la construction EXIT, c'est-à-dire que la commande doit passer à l'énoncé suivant, après le premier caractère de fin de la boucle (END_FOR, END_WHILE, ou END_REPEAT), à la suite de l'énoncé EXIT. Par exemple, après avoir exécuté les énoncés illustrés à la figure 22, la valeur de la variable SUM doit être 15, si la valeur de la variable booléenne FLAG est 0, et 6 si FLAG=1.

```
SUM := 0 ;
FOR I := 1 TO 3 DO
  FOR J := 1 TO 2 DO
    IF FLAG THEN EXIT ; END_IF
    SUM := SUM + J ;
  END_FOR ;
  SUM := SUM + I ;
END_FOR ;
```

Figure 22 – Exemple d'énoncé EXIT

L'énoncé FOR indique qu'une suite d'énoncés doit être exécutée de manière répétitive jusqu'à ce que le mot clé END_FOR soit atteint, une suite de valeurs progressives étant affectée à la variable de commande de la boucle FOR. La variable de commande, la valeur initiale et la valeur finale doivent être des expressions du même type entier (SINT, INT ou DINT) et ne doivent pas être modifiées par un des énoncés faisant l'objet de la répétition. L'énoncé FOR augmente ou diminue la variable de commande d'une valeur initiale jusqu'à une valeur finale par incréments déterminés par la valeur d'une expression; par défaut, cette valeur est 1. L'essai relatif à la condition terminale est effectué au début de chaque itération, de sorte que la suite d'énoncés n'est pas exécutée si la valeur initiale est supérieure à la valeur finale. La valeur de la variable de commande, après réalisation de la boucle FOR, dépend de l'application.

The IF statement specifies that a group of statements is to be executed only if the associated Boolean expression evaluates to the value 1 (true). If the condition is false, then either no statement is to be executed, or the statement group following the ELSE keyword (or the ELSIF keyword if its associated Boolean condition is true) is to be executed.

The CASE statement consists of an expression which shall evaluate to a variable of type INT (the "selector"), and a list of statement groups, each group being labelled by one or more integers or ranges of integer values. It specifies that the first group of statements, one of whose ranges contains the computed value of the selector, shall be executed. If the value of the selector does not occur in a range of any case, the statement sequence following the keyword ELSE (if it occurs in the CASE statement) shall be executed. Otherwise, none of the statement sequences shall be executed.

3.3.2.4 *Iteration statements*

Iteration statements specify that the group of associated statements shall be executed repeatedly. The FOR statement is used if the number of iterations can be determined in advance; otherwise, the WHILE or REPEAT constructs are used.

The EXIT statement shall be used to terminate iterations before the termination condition is satisfied.

When the EXIT statement is located within nested iterative constructs, exit shall be from the innermost loop in which the EXIT is located, that is, control shall pass to the next statement after the first loop terminator (END_FOR, END_WHILE, or END_REPEAT) following the EXIT statement. For instance, after executing the statements shown in figure 22, the value of the variable SUM shall be 15 if the value of the Boolean variable FLAG is 0, and 6 if FLAG=1.

```
SUM := 0 ;
FOR I := 1 TO 3 DO
   FOR J := 1 TO 2 DO
      IF FLAG THEN EXIT ; END_IF
      SUM := SUM + J ;
   END_FOR ;
   SUM := SUM + I ;
END_FOR ;
```

Figure 22 – EXIT statement example

The FOR statement indicates that a statement sequence shall be repeatedly executed, up to the END_FOR keyword, while a progression of values is assigned to the FOR loop control variable. The control variable, initial value, and final value shall be expressions of the same integer type (SINT, INT, or DINT) and shall not be altered by any of the repeated statements. The FOR statement increments the control variable up or down from an initial value to a final value in increments determined by the value of an expression; this value defaults to 1. The test for the termination condition is made at the beginning of each iteration, so that the statement sequence is not executed if the initial value exceeds the final value. The value of the control variable after completion of the FOR loop is implementation-dependent.

Un exemple de l'utilisation de l'énoncé FOR est donné dans la caractéristique n° 6 du tableau 56. Dans cet exemple, la boucle FOR est utilisée pour déterminer l'indice J de la première occurrence (le cas échéant) de la chaîne 'KEY' dans les éléments portant un nombre impair d'un tableau de chaînes WORDS et dotés d'une plage d'indice de (1..100). En l'absence d'occurrence, J aura la valeur 101.

L'énoncé WHILE déclenche l'exécution répétée de la suite d'énoncés s'étendant jusqu'au mot clé END_WHILE, jusqu'à ce que l'expression booléenne associée soit fausse. Si, à l'origine, l'expression est fausse, le groupe d'énoncés n'est pas exécuté du tout. Par exemple, la boucle FOR...END_FOR donnée dans le tableau 56 peut être réécrite à l'aide de la syntaxe WHILE...END_WHILE donnée dans le tableau 56.

L'énoncé REPEAT entraîne l'exécution répétée (et au moins une fois) de la suite d'énoncés s'étendant jusqu'au mot clé UNTIL, jusqu'à ce que la condition booléenne associée soit vraie. Ainsi, l'exemple WHILE...END_WHILE donné dans le tableau 56 peut être réécrit à l'aide de la syntaxe REPEAT...END_REPEAT donnée dans le tableau 56.

Les énoncés WHILE et REPEAT ne doivent pas être utilisés pour réaliser une synchronisation entre processus, par exemple en mettant en oeuvre une "boucle d'attente" affectée d'une condition terminale fixée de manière externe. Les éléments de SFC définis en 2.6 doivent être utilisés à cet effet.

Si un énoncé WHILE ou REPEAT est utilisé dans un algorithme pour lequel la satisfaction à la condition de terminaison de la boucle ou l'exécution d'un énoncé EXIT ne peut être garanti, cela doit être considéré comme une erreur au sens de 1.5.1.

4 Langages graphiques

Les langages graphiques définis dans la présente norme sont les langages LD (langage à contacts) et FBD (langage en blocs fonctionnels). Les éléments du "diagramme fonctionnel en séquence" définis en 2.6 peuvent être utilisés associés à l'un ou l'autre de ces langages.

4.1 *Eléments communs*

Les éléments définis au présent paragraphe s'appliquent aux deux langages graphiques de la présente norme, LD (langage à contacts) et FBD (Langage en blocs fonctionnels) ainsi qu'à la représentation graphique des éléments "diagramme fonctionnel en séquence" (SFC).

4.1.1 *Représentation des lignes et des blocs*

Les éléments de langage graphique définis au présent paragraphe sont dessinés à l'aide d'éléments linéaires utilisant des caractères qui appartiennent au jeu de caractères ISO/IEC 646, ou en utilisant des éléments graphiques ou semi-graphiques, comme illustré dans le tableau 57.

Les lignes peuvent être étendues par l'utilisation de *connecteurs,* comme illustré dans le tableau 57. Aucune mémorisation de données, ni association avec des éléments de données ne doit être associée à l'utilisation de connecteurs; par conséquent, pour éviter toute ambiguïté, ce doit être une *erreur* d'utiliser comme étiquette de connecteur un identificateur identique au nom d'un autre élément nommé dans la même unité d'organisation de programme.

An example of the usage of the FOR statement is given in feature 6 of table 56. In this example, the FOR loop is used to determine the index J of the first occurrence (if any) of the string 'KEY' in the odd-numbered elements of an array of strings WORDS with a subscript range of (1..100). If no occurrence is found, J will have the value 101.

The WHILE statement causes the sequence of statements up to the END_WHILE keyword to be executed repeatedly until the associated Boolean expression is false. If the expression is initially false, then the group of statements is not executed at all. For instance, the FOR...END_FOR example given in table 56 can be rewritten using the WHILE...END_WHILE construction shown in table 56.

The REPEAT statement causes the sequence of statements up to the UNTIL keyword to be executed repeatedly (and at least once) until the associated Boolean condition is true. For instance, the WHILE...END_WHILE example given in table 56 can be rewritten using the REPEAT...END_REPEAT construction shown in table 56.

The WHILE and REPEAT statements shall not be used to achieve interprocess synchronization, for example as a "wait loop" with an externally determined termination condition. The SFC elements defined in 2.6 shall be used for this purpose.

It shall be an *error* in the sense of 1.5.1 if a WHILE or REPEAT statement is used in an algorithm for which satisfaction of the loop termination condition or execution of an EXIT statement cannot be guaranteed.

4 Graphic languages

The graphic languages defined in this standard are LD (Ladder Diagram) and FBD (Function Block Diagram). The sequential function chart (SFC) elements defined in 2.6 can be used in conjunction with either of these languages.

4.1 *Common elements*

The elements defined in this clause apply to both the graphic languages in this Standard, that is, LD (Ladder Diagram) and FBD (Function Block Diagram), and to the graphic representation of sequential function chart (SFC) elements.

4.1.1 *Representation of lines and blocks*

The graphic language elements defined in this clause are drawn with line elements using characters from the ISO/IEC 646 character set, or using graphic or semigraphic elements, as shown in table 57.

Lines can be extended by the use of *connectors* as shown in table 57. No storage of data or association with data elements shall be associated with the use of connectors; hence, to avoid ambiguity, it shall be an *error* if the identifier used as a connector label is the same as the name of another named element within the same program organization unit.

4.1.2 *Sens du flux dans les réseaux*

Un *réseau* est défini comme un ensemble maximal d'éléments graphiques connectés entre eux, à l'exclusion des barres d'alimentation droite et gauche dans le cas des réseaux en langage LD défini en 4.2. On doit veiller à associer, à chaque réseau ou groupe de réseaux d'un langage graphique, une *étiquette de réseau* délimitée sur sa droite par deux points (:). Cette étiquette doit avoir la forme d'un identificateur ou d'un entier décimal non signé, conformément à l'article 2 de la présente partie. Le *domaine* d'un réseau et son étiquette doivent être locaux à l'unité d'organisation de programme dans laquelle le réseau se situe. Des exemples de réseaux et d'étiquettes de réseaux sont fournis dans l'annexe F.

Les langages graphiques sont utilisés pour représenter le cheminement d'une quantité conceptuelle à travers un ou plusieurs réseaux représentant un plan de commande, c'est-à-dire:

– "flux d'énergie", analogue à l'écoulement de l'énergie électrique dans un système à relais électromécaniques, généralement utilisé dans les schémas à contacts;

– "flux des signaux", analogue à l'écoulement des signaux entre les éléments d'un système de traitement du signal, généralement utilisé dans les schémas en blocs fonctionnels;

– "flux d'activité", analogue au cheminement des commandes entre les éléments d'une organisation ou entre les phases d'un séquenceur électromécanique, généralement utilisé dans les schémas de diagramme fonctionnel en séquence.

La quantité conceptuelle appropriée doit circuler le long des lignes reliant les éléments d'un réseau, en respectant les règles suivantes:

1) La circulation d'énergie en langage LD doit se faire de la gauche vers la droite.

2) Le flux des signaux en langage FBD doit s'effectuer de la sortie (à droite) d'une fonction ou d'un bloc fonctionnel vers l'entrée (à gauche) de la fonction ou du ou des blocs fonctionnels raccordés.

3) Le flux d'activité entre les éléments de diagramme fonctionnel en séquence (SFC) définis en 2.6 doit se faire du bas d'une étape vers le haut de l'étape ou des étapes suivantes correspondantes grâce à la transition appropriée.

4.1.2 *Direction of flow in networks*

A *network* is defined as a maximal set of interconnected graphic elements, excluding the left and right rails in the case of networks in the LD language defined in 4.2. Provision shall be made to associate with each network or group of networks in a graphic language a *network label* delimited on the right by a colon (:). This label shall have the form of an identifier or an unsigned decimal integer as defined in clause 2 of this part. The *scope* of a network and its label shall be *local* to the program organization unit in which the network is located. Examples of networks and network labels are shown in annex F.

Graphic languages are used to represent the flow of a conceptual quantity through one or more networks representing a control plan, that is:

– "Power flow", analogous to the flow of electric power in an electromechanical relay system, typically used in relay ladder diagrams;

– "Signal flow", analogous to the flow of signals between elements of a signal processing system, typically used in function block diagrams;

– "Activity flow", analogous to the flow of control between elements of an organization, or between the steps of an electromechanical sequencer, typically used in sequential function charts.

The appropriate conceptual quantity shall flow along lines between elements of a network according to the following rules:

1) Power flow in the LD language shall be from left to right.

2) Signal flow in the FBD language shall be from the output (right-hand) side of a function block to the input (left-hand) side of the function or function block(s) so connected.

3) Activity flow between the SFC elements defined in 2.6 shall be from the bottom of a step through the appropriate transition to the top of the corresponding successor step(s).

Tableau 57 – Représentation des lignes et des blocs

N°	Caractéristique	Exemple
1 2	Lignes horizontales: Caractère "moins" ISO/IEC 646 Graphique ou semi-graphique	`-----`
3 4	Lignes verticales: Caractère "ligne verticale" ISO/IEC 646 Graphique ou semi-graphique	`|`

5
6
	Jonction ligne horizontale/ligne verticale:
	Caractère "plus" ISO/IEC 646
	Graphique ou semi-graphique
		`  |`
		`--+--`
		`  |`

7
8
	Croisements de lignes sans jonction:
	Caractères ISO/IEC 646
	Graphique ou semi-graphique
		`  |  |`
		`--------|----`
		`  |  |`

9
10
	Coins connectés et non connectés:
	Caractères ISO/IEC 646
	Graphique ou semi-graphique
		`  |  |`
		`----+  +----`
		`  |`
		`----+-+  +----`
		`  |  |  |`

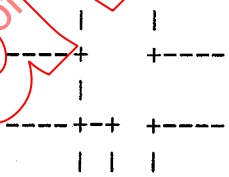
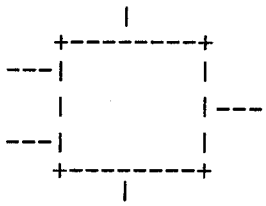
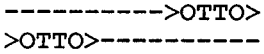
11
12
	Blocs avec lignes connectées:
	Caractères ISO/IEC 646
	Graphique ou semi-graphique
		`  |`
		`  +--------+`
		`---|  |`
		`  |  |---`
		`---|  |`
		`  +--------+`
		`  |`

13 14	Connecteurs utilisant des caractères ISO/IEC 646: Connecteur Suite d'une ligne connectée Connecteurs graphiques ou semi-graphiques	`---------->OTTO>` `>OTTO>----------`

4.1.3 *Evaluation de réseaux*

L'ordre dans lequel les réseaux et leurs éléments sont évalués n'est pas nécessairement l'ordre dans lequel ils sont étiquetés ou affichés. De même, il n'est pas impératif que tous les réseaux soient évalués avant que l'évaluation d'un réseau donné puisse être répétée. Cependant, lorsque le corps d'une unité d'organisation de programme se compose d'un ou de plusieurs réseaux, les résultats de l'évaluation de réseaux à l'intérieur de ce corps doivent être fonctionnellement équivalents au respect des règles suivantes:

Table 57 – Representation of lines and block

No.	Feature	Example
1 2	Horizontal lines: ISO/IEC 646 "minus" character Graphic or semigraphic	
3 4	Vertical lines: ISO/IEC 646 "vertical line" character Graphic or semigraphic	
5 6	Horizontal/vertical connection: ISO/IEC 646 "plus" character Graphic or semigraphic	
7 8	Line crossings without connection: ISO/IEC 646 characters Graphic or semigraphic	
9 10	Connected and non-connected corners: ISO/IEC 646 characters Graphic or semigraphic	
11 12	Blocks with connecting lines: ISO/IEC 646 characters Graphic or semigraphic	
13 14	Connectors using ISO/IEC 646 characters: Connector Continuation of a connected line Graphic or semigraphic connectors	

4.1.3 *Evaluation of networks*

The order in which networks and their elements are evaluated is not necessarily the same as the order in which they are labelled or displayed. Similarly, it is not necessary that all networks be evaluated before the evaluation of a given network can be repeated. However, when the body of a program organization unit consists of one or more networks, the results of network evaluation within said body shall be functionally equivalent to the observance of the following rules:

1) Aucun élément d'un réseau ne doit être évalué tant que les états de toutes ses entrées n'ont pas été évalués.

2) L'évaluation d'un élément de réseau ne doit être terminée que lorsque les états de toutes ses sorties auront été évalués.

3) L'évaluation d'un élément de réseau n'est terminée que lorsque les sorties de tous ses éléments ont été évaluées, même si le réseau contient l'un des éléments de commande et d'exécution définis en 4.1.4.

4) L'ordre dans lequel les réseaux sont évalués doit être conforme aux exigences de 4.2.6 en ce qui concerne le langage LD, et aux exigences de 4.3.3 pour le langage FBD.

On dit qu'il existe dans un réseau un *chemin d'asservissement* lorsque la sortie d'une fonction ou d'un bloc fonctionnel est utilisée comme une entrée de fonction ou de bloc fonctionnel qui le précède dans le réseau; la variable associée est appelée *variable d'asservissement*. Par exemple, la variable booléenne RUN est la variable d'asservissement dans l'exemple de la figure 23. Une variable d'asservissement peut également être un élément de sortie d'une structure de données de bloc fonctionnel telle que définie en 2.5.2.

Les chemins d'asservissement peuvent être utilisés dans les langages graphiques définis en 4.2 et 4.3, sous réserve d'observer les règles suivantes:

1) Les boucles explicites, telles que celle représentée par la figure 23a ne doivent apparaître que dans le langage FBD défini en 4.3.

2) L'utilisateur doit avoir la possibilité de définir l'ordre d'exécution des éléments dans une boucle explicite, par exemple, en sélectionnant des variables d'asservissement de manière à former une boucle implicite, comme l'illustre la figure 23b.

3) Les variables d'asservissement doivent être initialisées en utilisant l'un des mécanismes définis à l'article 2. La valeur initiale doit être utilisée au cours de la première évaluation du réseau.

4) Une fois que l'élément doté d'une variable d'asservissement comme sortie a été évalué, la nouvelle valeur de la variable d'asservissement doit être utilisée jusqu'à la prochaine évaluation de cet élément.

1) No element of a network shall be evaluated until the states of all of its inputs have been evaluated.

2) The evaluation of a network element shall not be complete until the states of all of its outputs have been evaluated.

3) The evaluation of a network is not complete until the outputs of all of its elements have been evaluated, even if the network contains one of the execution control elements defined in 4.1.4.

(4) The order in which networks are evaluated shall conform to the provisions of 4.2.6 for the LD language and 4.3.3 for the FBD language.

A *feedback path* is said to exist in a network when the output of a function or function block is used as the input to a function or function block which precedes it in the network; the associated variable is called *a feedback variable*. For instance, the Boolean variable RUN is the feedback variable in the example shown in figure 23. A feedback variable can also be an output element of a function block data structure as defined in 2.5.2.

Feedback paths can be utilized in the graphic languages defined in 4.2 and 4.3, subject to the following rules:

1) Explicit loops such as the one shown in 23a shall only appear in the FBD language defined in 4.3.

2) It shall be possible for the user to define the order of execution of the elements in an explicit loop, for instance by selection of feedback variables to form an implicit loop as shown in figure 23b.

3) Feedback variables shall be initialized by one of the mechanisms defined in clause 2. The initial value shall be used during the first evaluation of the network.

4) Once the element with a feedback variable as output has been evaluated, the new value of the feedback variable shall be used until the next evaluation of the element.

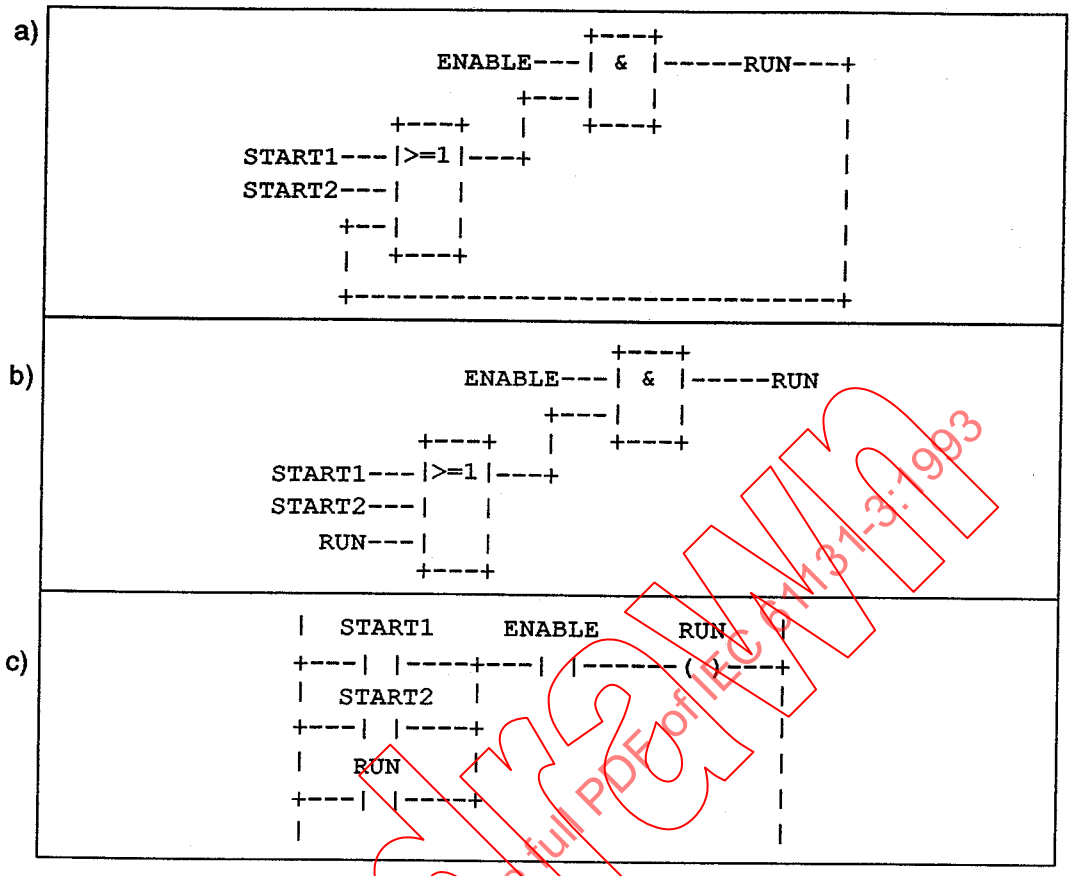


Figure 23 – Exemple de chemin d'asservissement
a) Boucle explicite
b) Boucle implicite
c) Equivalent en langage LD

4.1.4 *Eléments de commande d'exécution*

Le transfert de la commande des programmes en langages LD et FBD doit être représenté par les éléments graphiques du tableau 58.

Les sauts doivent être représentés par une ligne de signal booléenne se terminant par une double flèche. La ligne de signal pour un saut doit partir d'une variable booléenne, de la sortie booléenne d'une fonction ou d'un bloc fonctionnel ou de la ligne de flux d'énergie d'un schéma à contacts. Un transfert de commande des programmes vers l'étiquette de réseau désignée doit s'effectuer lorsque la valeur booléenne de la ligne de signal est 1 (TRUE); par conséquent, le saut inconditionnel est un cas particulier du saut conditionnel.

La cible d'un saut doit être une étiquette de réseau à l'intérieur de l'unité d'organisation de programme où se produit le saut. Si le saut intervient dans une instruction ACTION...END_ACTION, la cible du saut doit se trouver à l'intérieur de la même instruction.

Les retours conditionnels de fonctions et de blocs fonctionnels doivent être mis en oeuvre à l'aide d'une instruction RETURN, comme l'indique le tableau 58. L'exécution du programme doit être renvoyée à l'entité appelante lorsque l'entrée booléenne a la valeur 1 (TRUE), et doit se poursuivre normalement lorsque l'entrée booléenne a la valeur 0 (FALSE). Les retours inconditionnels doivent être assurés à la fin physique de la fonction ou du bloc fonctionnel, ou grâce à un élément RETURN connecté à la barre d'alimentation gauche du langage LD, comme l'indique le tableau 58.

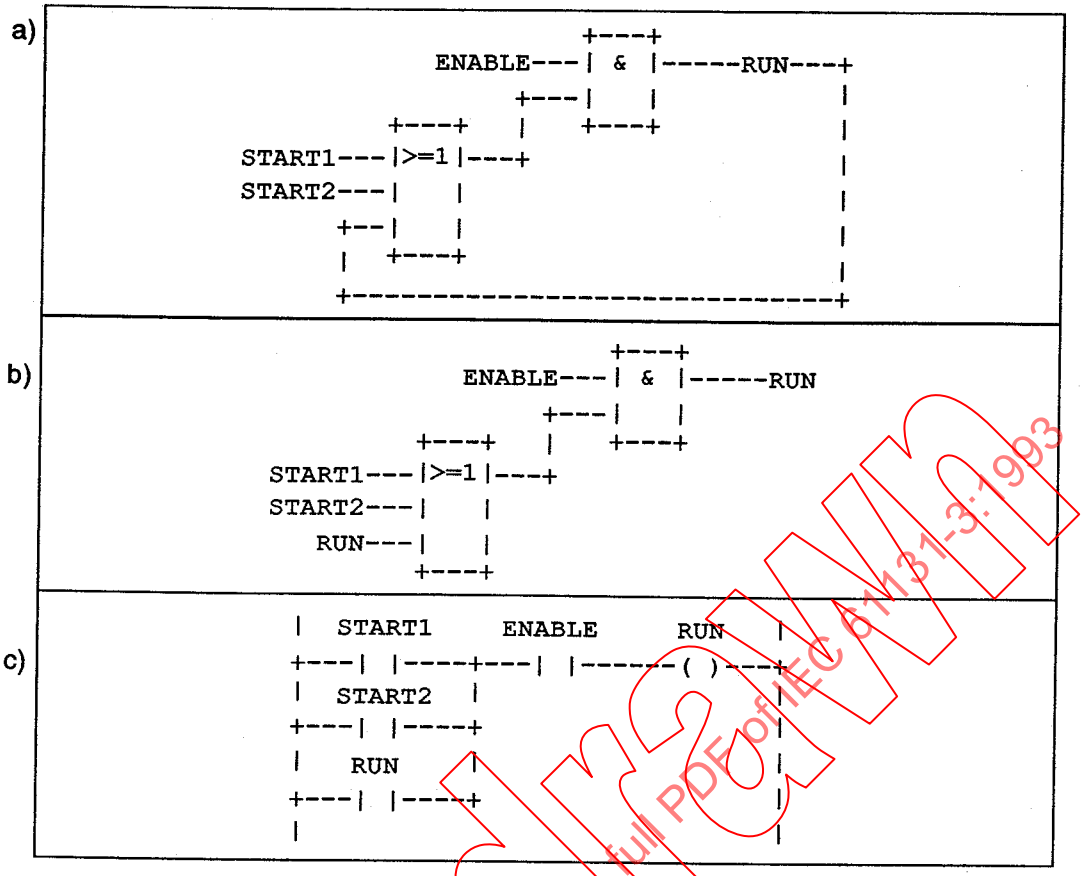


Figure 23 – Feedback path example
a) Explicit loop
b) Implicit loop
c) LD language equivalent

4.1.4 *Execution control elements*

Transfer of program control in the LD and FBD languages shall be represented by the graphical elements shown in table 58.

Jumps shall be shown by a Boolean signal line terminated in a double arrowhead. The signal line for a jump condition shall originate at a Boolean variable, at a Boolean output of a function or function block, or on the power flow line of a ladder diagram. A transfer of program control to the designated network label shall occur when the Boolean value of the signal line is 1 (TRUE); thus, the unconditional jump is a special case of the conditional jump.

The target of a jump shall be a network label within the program organization unit within which the jump occurs. If the jump occurs within an ACTION...END_ACTION construct, the target of the jump shall be within the same construct.

Conditional returns from functions and function blocks shall be implemented using a RETURN construction as shown in table 58. Program execution shall be transferred back to the invoking entity when the Boolean input is 1 (TRUE), and shall continue in the normal fashion when the Boolean input is 0 (FALSE). Unconditional returns shall be provided by the physical end of the function or function block, or by a RETURN element connected to the left rail in the LD language, as shown in table 58.

Tableau 58 – Eléments de commande d'exécution graphiques

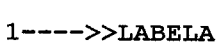
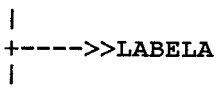
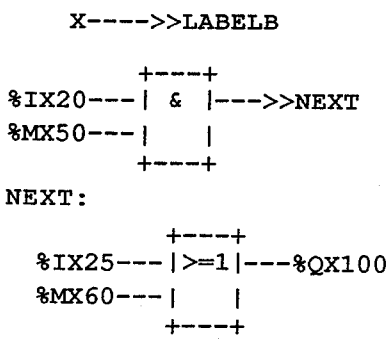
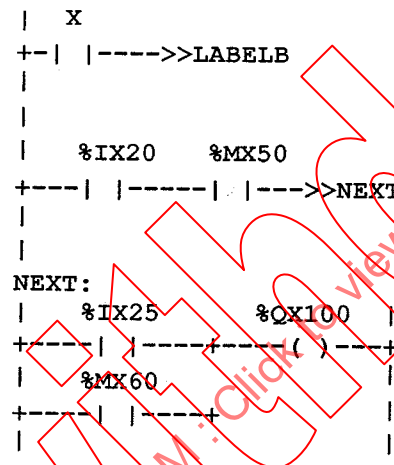
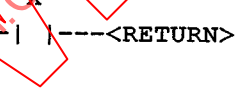
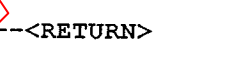
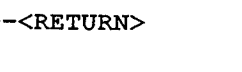
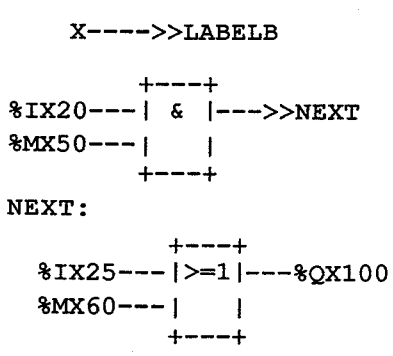
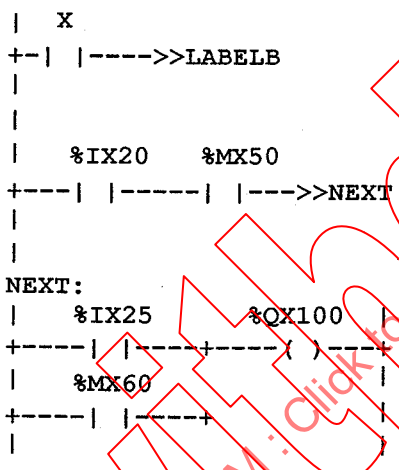
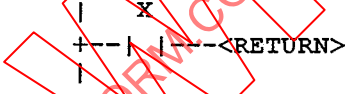
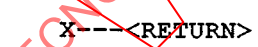
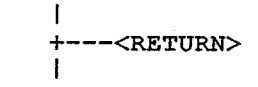
N°	Symbole/exemple	Signification
1		Saut inconditionnel: Langage FBD
2		Langage LD
3		Saut conditionnel (Langage FBD) Exemple: Condition de saut Cible du saut
4		Saut conditionnel (Langage LD) Exemple: Condition de saut Cible du saut
5		Retour conditionnel: Langage LD
6		Langage FBD
7	END_FUNCTION END_FUNCTION_BLOCK	Retour inconditionnel: d'une FUNCTION d'un FUNCTION_BLOCK
8		Autres représentations possibles en langage LD

Table 58 – Graphic execution control elements

No.	Symbol/Example	Explanation
1		Unconditional Jump: FBD Language
2		LD language
3		Conditional Jump (FBD Language) Example: Jump Condition Jump Target
4		Conditional Jump (LD Language) Example: Jump Condition Jump Target
5		Conditional Return: LD Language
6		FBD Language
7	END_FUNCTION END_FUNCTION_BLOCK	Unconditional Return from FUNCTION from FUNCTION_BLOCK
8		Alternative representation in LD language

4.2 *Langage à contacts (LD)*

Ce paragraphe définit le langage LD pour la programmation, par langage à contacts, des automates programmables.

Un programme en langage LD permet à l'automate programmable de tester et de modifier des données à l'aide de symboles graphiques standardisés. Ces symboles sont organisés en réseaux de la même manière que les "échelons" d'un schéma logique à contacts. Les réseaux LD sont reliés sur la gauche et sur la droite par des *barres d'alimentation.*

4.2.1 *Barres d'alimentation*

Un réseau LD doit être délimité sur la gauche par une ligne verticale connue sous le nom de *barre d'alimentation gauche,* et sur la droite par une autre ligne verticale connue sous le nom de *barre d'alimentation droite.* La barre d'alimentation droite peut être explicite ou implicite. Voir tableau 59.

Tableau 59 – Barres d'alimentation

N°	Symbole	Description
1		Barre d'alimentation gauche (avec liaison horizontale attachée)
2		Barre d'alimentation droite (avec liaison horizontale attachée)

4.2.2 *Eléments de liaison et états*

Les éléments de liaison peuvent être horizontaux ou verticaux comme figurés dans le tableau 60. L'état de l'élément de liaison doit être désigné par "ON" ou "OFF" correspondant respectivement aux valeurs booléennes littérales 1 ou 0. Le terme *état de la liaison* doit être synonyme du terme *flux d'énergie.*

L'état de la barre d'alimentation gauche doit être considéré comme ON à tout moment. Aucun état n'est défini pour la barre d'alimentation droite.

Un élément de liaison horizontale doit être signalé par une ligne horizontale. Un élément de liaison horizontale transmet l'état de l'élément situé immédiatement à sa gauche à l'élément situé immédiatement à sa droite.

L'élément de liaison verticale doit être composé d'une ligne verticale croisant un ou plusieurs éléments de liaison horizontale de chaque côté. L'état de la liaison verticale doit représenter le OU logique des états ON des liaisons horizontales situées à sa gauche, c'est-à-dire que l'état de la liaison verticale doit être:

4.2 *Language LD (Ladder Diagram)*

This subclause defines the LD language for ladder diagram programming of programmable controllers.

A LD program enables the programmable controller to test and modify data by means of standardized graphic symbols. These symbols are laid out in networks in a manner similar to a "rung" of a relay ladder logic diagram. LD networks are bounded on the left and right by *power rails*.

4.2.1 *Power rails*

As shown in table 59, LD network shall be delimited on the left by a vertical line known as the *left power rail*, and on the right by a vertical line known as the *right power rail*. The right power rail may be explicit or implied.

Table 59 – Power rails

No.	Symbol	Description
1		Left power rail (with attached horizontal link)
2		Right power rail (with attached horizontal link)

4.2.2 *Link elements and states*

As shown in table 60, link elements may be horizontal or vertical. The state of the link element shall be denoted "ON" or "OFF", corresponding to the literal Boolean values 1 or 0, respectively. The term *link state* shall be synonymous with the term *power flow.*

The state of the left rail shall be considered ON at all times. No state is defined for the right rail.

A horizontal link element shall be indicated by a horizontal line. A horizontal link element transmits the state of the element on its immediate left to the element on its immediate right.

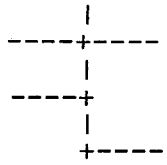
The vertical link element shall consist of a vertical line intersecting with one or more horizontal link elements on each side. The state of the vertical link shall represent the inclusive OR of the ON states of the horizontal links on its left side, that is, the state of the vertical link shall be:

– OFF, si les états de toutes les liaisons horizontales attachées du côté gauche sont OFF;

– ON, si l'état d'au moins une des liaisons horizontales attachées du côté gauche est ON.

L'état de la liaison verticale doit être copié sur toutes les liaisons horizontales qui lui sont rattachées du côté droit. L'état de la liaison verticale ne doit être copié sur aucune des liaisons horizontales qui lui sont rattachées du côté gauche.

Tableau 60 – Eléments de liaison

N°	Symbole	Description
1	----------	Liaison horizontale
2		Liaison verticale (avec liaisons horizontales attachées)

4.2.3 *Contacts*

Un *contact* est un élément qui attribue à la liaison horizontale située sur sa droite, un état qui est le résultat du AND booléen de l'état de la liaison horizontale située sur sa gauche avec une fonction appropriée d'une entrée, d'une sortie ou d'une variable mémoire booléenne associée. Un contact ne modifie pas la valeur de la variable booléenne associée. Le tableau 61 présente les symboles normalisés de contacts.

4.2.4 *Bobinages*

Un *bobinage* copie l'état de la liaison située à sa gauche sur la liaison située à sa droite sans aucune modification et mémorise une fonction appropriée de l'état ou de la transition de la liaison gauche dans la variable booléenne associée. Le tableau 62 présente les symboles normalisés de bobinages.

4.2.5 *Fonctions et blocs fonctionnels*

La représentation des fonctions et des blocs fonctionnels dans le langage LD doit être conforme à la définition de l'article 2 de la présente partie, excepté dans les cas suivants:

1) les connexions de paramètres réels peuvent éventuellement être représentées en écrivant les données ou la variable appropriées à l'extérieur du pavé, à proximité du nom du paramètre formel situé à l'intérieur;

2) pour que l'énergie puisse circuler dans un bloc, il faut au minimum une entrée booléenne et une sortie booléenne sur ce bloc.

4.2.6 *Ordre d'évaluation des réseaux*

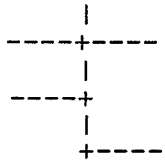
Dans une unité d'organisation de programme écrite en langage LD, les réseaux doivent être évalués dans l'ordre de leur apparition, de haut en bas, dans le schéma à contacts, sauf dans le cas où cet ordre est modifié par les éléments de commande d'exécution définis en 4.1.4.

– OFF if the states of all the attached horizontal links to its left are OFF;

– ON if the state of one or more of the attached horizontal links to its left is ON.

The state of the vertical link shall be copied to all of the attached horizontal links on its right. The state of the vertical link shall not be copied to any of the attached horizontal links on its left.

Table 60 – Link elements

No.	Symbol	Description
1	----------	Horizontal link
2		Vertical link (with attached horizontal links)

4.2.3 *Contacts*

A *contact* is an element which imparts a state to the horizontal link on its right side which is equal to the Boolean AND of the state of the horizontal link at its left side with an appropriate function of an associated Boolean input, output, or memory variable. A contact does not modify the value of the associated Boolean variable. Standard contact symbols are given in table 61.

4.2.4 *Coils*

A *coil* copies the state of the link on its left to the link on its right without modification, and stores an appropriate function of the state or transition of the left link into the associated Boolean variable. Standard coil symbols are given in table 62.

4.2.5 *Functions and function blocks*

The representation of functions and function blocks in the LD language shall be as defined in clause 2 of this part, with the following exceptions:

1) Actual parameter connections may optionally be shown by writing the appropriate data or variable outside the block adjacent to the formal parameter name on the inside.

2) At least one Boolean input and one Boolean output shall be shown on each block to allow for power flow through the block.

4.2.6 *Order of network evaluation*

Within a program organization unit written in LD, networks shall be evaluated in top-to-bottom order as they appear in the ladder diagram, except as this order is modified by the execution control elements defined in 4.1.4.

Tableau 61 – Contacts

Contacts statiques		
N°	Symbole	Description
1 2	*** --| |-- ou *** --! !--	*Contact normalement au repos* L'état de la liaison gauche est copié sur la liaison droite si l'état de la variable booléenne associée (désignée par "***") est ON. Dans tous les autres cas, l'état de la liaison droite est OFF.
3 4	*** --|/|-- ou *** --!/!--	*Contact normalement au travail* L'état de la liaison gauche est copié sur la liaison droite si l'état de la variable booléenne associée (désignée par "***") est OFF. Dans tous les autres cas, l'état de la liaison droite est OFF.
Contacts détecteurs de transition		
5 6	*** --|P|-- ou *** --!P!--	*Contact détecteur de transition positive* L'état de la liaison droite est ON d'une évaluation de cet élément jusqu'à l'évaluation suivante, lorsqu'une transition de OFF à ON de la variable associée est détectée, alors que la liaison gauche est à l'état ON. Dans tous les autres cas, l'état de la liaison droite doit être OFF.
7 8	*** --|N|-- ou *** --!N!--	*Contact détecteur de transition négative* L'état de la liaison droite est ON d'une évaluation de cet élément jusqu'à l'évaluation suivante, lorsqu'une transition de ON à OFF de la variable associée est détectée, alors que la liaison gauche est à l'état ON. Dans tous les autres cas, l'état de la liaison droite doit être OFF.
NOTE - Comme spécifié en 2.1.1, le point d'exclamation "!" doit être utilisé lorsque la variante nationale du jeu de caractères ne comporte pas la barre verticale "|".		

Table 61 – Contacts

No.	Symbol	Description
Static contacts		
1 2	*** --| |-- or *** --! !--	*Normally open contact* The state of the left link is copied to the right link if the state of the associated Boolean variable (indicated by "***") is ON. Otherwise, the state of the right link is OFF.
3 4	*** --|/|-- or *** --!/!--	*Normally closed contact* The state of the left link is copied to the right link if the state of the associated Boolean variable is OFF. Otherwise, the state of the right link is OFF.
Transition-sensing contacts		
5 6	*** --|P|-- or *** --!P!--	*Positive transition-sensing contact* The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from OFF to ON is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.
7 8	*** --|N|-- or *** --!N!--	*Negative transition-sensing contact* The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from ON to OFF is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.
NOTE - As specified in 2.1.1, the exclamation mark "!" shall be used when a national character set does not support the vertical bar "|".		

Tableau 62 – Bobinages

Bobinages fugitifs		
N°	Symbole	Description
1	*** --( )--	*Bobinage* L'état de la liaison gauche est copié dans la variable booléenne associée et vers la liaison droite.
2	*** --(/)--	*Bobinage "négativé"* L'état de la liaison gauche est copié vers la liaison droite. L'inverse de l'état de la liaison gauche est copié dans la variable booléenne associée, c'est-à-dire que, si par exemple, l'état de la liaison gauche est OFF, l'état de la variable associée est ON, et vice versa.
Bobinages à maintien		
3	*** --(S)--	*Bobinage SET (verrouillage)* La variable booléenne associée est mise à l'état ON lorsque la liaison gauche est à l'état ON, et elle conserve cet état jusqu'à remise à zéro par le bobinage RESET.
4	*** --(R)--	*Bobinage RESET (déverrouillage)* La variable booléenne associée est remise à l'état OFF lorsque la liaison gauche est à l'état ON, et elle conserve cet état jusqu'à son positionnement par un bobinage SET.
Bobinages à mémorisation (voir note)		
5	*** ----(M)----	Bobinage de mémorisation
6	*** ----(SM)----	Bobinage de mémorisation SET
7	*** ----(RM)----	Bobinage de mémorisation RESET
Bobinage détecteurs de transition		
8	*** --(P)--	*Bobinage détecteur de transition positive* L'état de la variable booléenne associée est ON d'une évaluation de cet élément jusqu'à l'évaluation suivante, lorsqu'une transition de OFF à ON de la liaison gauche est détectée. L'état de la liaison gauche est toujours copié sur la liaison droite.
9	*** --(N)--	*Bobinage détecteur de transition négative* L'état de la variable booléenne associée est ON de l'évaluation de cet élément jusqu'à l'évaluation suivante, lorsqu'une transition de ON à OFF de la liaison gauche est détectée. L'état de la liaison gauche est toujours copié sur la liaison droite.
NOTE - Les bobinages 5, 6 et 7 ont respectivement le même effet que les bobinages 1, 3 et 4, excepté que la variable booléenne associée est automatiquement déclarée être dans la mémoire non volatile, sans qu'il soit nécessaire d'utiliser explicitement la déclaration VAR RETAIN définie en 2.4.2.		

Table 62 – Coils

No.	Symbol	Description
		Momentary coils
1	*** --( )--	*Coil* The state of the left link is copied to the associated Boolean variable and to the right link.
2	*** --(/)--	*Negated coil* The state of the left link is copied to the right link. The inverse of the state of the left link is copied to the associated Boolean variable, that is, if the state of the left link is OFF, then the state of the associated variable is ON, and vice versa.
		Latched coils
3	*** --(S)--	*SET (latch) coil* The associated Boolean variable is set to the ON state when the left link is in the ON state, and remains set until reset by a RESET coil.
4	*** --(R)--	*RESET (unlatch) coil* The associated Boolean variable is reset to the OFF state when the left link is in the ON state, and remains reset until set by a SET coil.
		Retentive coils (see note)
5	*** ----(M)----	Retentive (Memory) coil
6	*** ----(SM)----	SET retentive (Memory) coil
7	*** ----(RM)----	RESET retentive (Memory) coil
		Transition-sensing coils
8	*** --(P)--	*Positive transition-sensing coil* The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from OFF to ON is sensed. The state of the left link is always copied to the right link.
9	*** --(N)--	*Negative transition-sensing coil* The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from ON to OFF is sensed. The state of the left link is always copied to the right link.

NOTE - The action of coils 5, 6, and 7 is identical to that of coils 1, 3, and 4, respectively, except that the associated Boolean variable is automatically declared to be in retentive memory without the explicit use of the VAR RETAIN declaration defined in 2.4.2.

4.3 *Langage FBD (langage en blocs fonctionnels)*

4.3.1 *Généralités*

Le présent paragraphe définit le langage FBD, un langage graphique utilisé pour la programmation d'automates programmables qui, dans la mesure du possible, est cohérent avec la CEI 617-12. En cas de désaccords entre la présente Norme et la Norme CEI 617, les règles édictées dans la présente norme doivent être appliquées à la programmation des automates programmables en langage FBD.

Les règles de 2 et 4.1 doivent être appliquées à la construction et à l'interprétation des programmes d'automates programmables en langage FBD.

Des exemples d'utilisation du langage FBD sont fournis dans l'annexe F.

4.3.2 *Combinaison d'éléments*

Les éléments du langage FBD doivent être connectés entre eux par des lignes de circulation des signaux, suivant les conventions de 4.1.2.

Les sorties des blocs fonctionnels ne doivent pas être connectées entre elles. En particulier, la construction "OR câblé" du langage LD n'est pas autorisée dans le langage FBD; il est impératif d'utiliser à la place un bloc "OR" booléen, comme indiqué dans la figure 24.

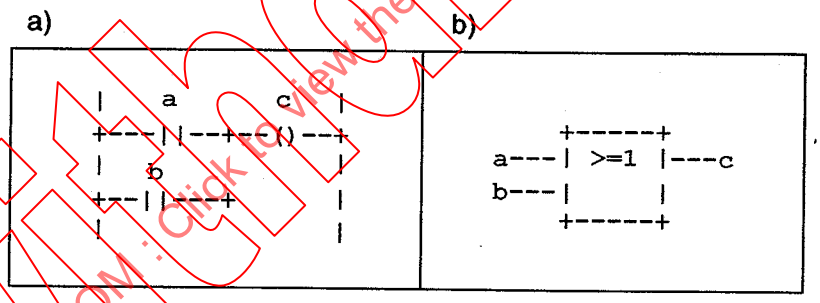


Figure 24 – Exemples de OR booléen
a) "OR câblé" en langage LD
b) Fonction en langage FBD

4.3.3 *Ordre d'évaluation des réseaux*

L'ordre d'évaluation des réseaux dans une unité d'organisation de programme écrite en langage FBD doit obéir à la règle suivante: l'évaluation d'un réseau doit être achevée avant que ne débute celle d'un autre réseau utilisant une ou plusieurs des sorties du réseau faisant l'objet de l'évaluation précédente.

4.3 *Language FBD (Function Block Diagram)*

4.3.1 *General*

This subclause defines FBD, a graphic language for the programming of programmable controllers which is consistent, as far as possible, with IEC 617-12. Where conflicts exist between this standard and IEC 617, the provisions of this standard shall apply for the programming of programmable controllers in the FBD language.

The provisions of clauses 2 and 4.1 shall apply to the construction and interpretation of programmable controller programs in the FBD language.

Examples of the use of the FBD language are given in annex F.

4.3.2 *Combination of elements*

Elements of the FBD language shall be interconnected by signal flow lines following the conventions of 4.1.2.

Outputs of function blocks shall not be connected together. In particular, the "wired-OR" construct of the LD language is not allowed in the FBD language; an explicit Boolean "OR" block is required instead, as shown in figure 24.

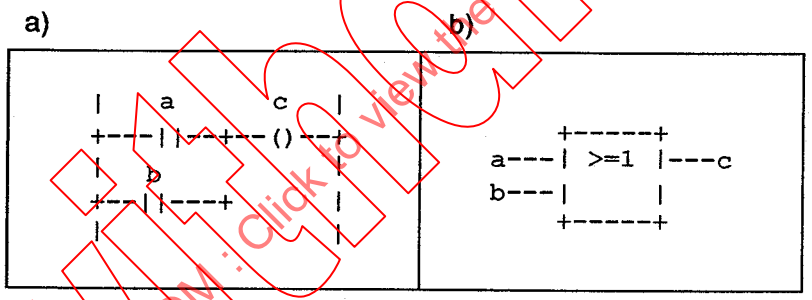


Figure 24 – Boolean OR examples
a) "Wired-OR" in LD language
b) Function in FBD language

4.3.3 *Order of network evaluation*

Within a program organization unit written in the FBD language, the order of network evaluation shall follow the rule that the evaluation of a network shall be complete before starting the evaluation of another network which uses one or more of the outputs of the preceding evaluated network.

Annexe A
(normative)

Méthode de spécification pour les langages littéraux

Les langages de programmation sont spécifiés en termes d'une *syntaxe,* qui définit les combinaisons de symboles autorisées pouvant être utilisées pour définir un programme, et d'une *sémantique* qui définit les relations existant entre les opérations programmées et les combinaisons de symboles définies par la syntaxe.

A.1 Syntaxe

Une syntaxe est définie par un ensemble de *symboles terminaux,* utilisés pour spécifier un programme, un ensemble de *symboles non terminaux,* définis en terme de symboles terminaux, et un ensemble de *règles de production* spécifiant ces définitions.

A.1.1 *Symboles terminaux*

Les symboles terminaux utilisés dans les programmes d'automates programmables littéraux doivent se composer de combinaisons des caractères appartenant au jeu ISO/IEC 646. Dans le cas d'échange de programmes entre systèmes, ces caractères doivent être représentés par les codes de caractères à 7 bits définis dans l'ISO/IEC 646.

Dans le cadre de cette partie, les symboles littéraux terminaux se composent du cordon de caractères approprié entre apostrophes ou entre guillemets. Par exemple, un symbole terminal composé du cordon de caractères ABC peut être représenté par:

"ABC"

ou par:

'ABC'

Cela permet de représenter des cordons contenant soit des apostrophes, soit des guillemets; par exemple, un symbole terminal consistant en un guillemet serait représenté par: '"'.

Le délimiteur de fin de ligne, représenté par le cordon de caractères EOL sans apostrophes ni guillemets, est un symbole terminal spécial utilisé dans cette syntaxe. Ce symbole doit normalement être composé du caractère CR (Retour Chariot) défini dans l'ISO/IEC 646. Tout écart par rapport à cette règle doit être signalé par les fabricants; dans tous les cas, seuls les caractères définis dans l'ISO/IEC 646 sont autorisés.

Un second symbole spécial utilisé dans cette syntaxe est le "cordon nul", c'est-à-dire un cordon ne contenant aucun caractère. Il est représenté par le symbole terminal NIL.

Annex A
(normative)

Specification method for textual languages

Programming languages are specified in terms of a *syntax*, which specifies the allowable combinations of symbols which can be used to define a program; and a set of *semantics*, which specify the relationship between programmed operations and the symbol combinations defined by the syntax.

A.1 Syntax

A syntax is defined by a set of *terminal symbols* to be utilized for program specification; a set of *non-terminal symbols* defined in terms of the terminal symbols; and a set of *production rules* specifying those definitions.

A.1.1 *Terminal symbols*

The terminal symbols for textual programmable controller programs shall consist of combinations of the characters in the ISO/IEC 646 character set. For interchange of programs between systems, these characters shall be represented by the seven-bit character codes defined in ISO 646.

For the purposes of this part, terminal textual symbols consist of the appropriate character string enclosed in paired single or double quotes. For example, a terminal symbol represented by the character string ABC can be represented by either

"ABC"

or

'ABC'

This allows the representation of strings containing either single or double quotes; for instance, a terminal symbol consisting of the double quote itself would be represented by '"'.

A special terminal symbol utilized in this syntax is the end-of-line delimiter, which is represented by the unquoted character string EOL. This symbol shall normally consist of the FE5 (CR = carriage return) character defined by ISO/IEC 646. Language implementors shall specify any deviation from this usage; in any case, no characters other than those in ISO/IEC 646 are allowed.

A second special terminal symbol utilized in this syntax is the "null string", that is, a string containing no characters. This is represented by the terminal symbol NIL.

A.1.2 *Symboles non terminaux*

Les symboles littéraux non terminaux doivent être représentés par des cordons composés de lettres minuscules, de chiffres et du caractère de soulignement (_); ces cordons doivent obligatoirement débuter par une lettre minuscule. Par exemple, les cordons:

nonterm1

et

non_term_2

sont des symboles non terminaux valides, alors que les cordons:

3nonterm

et

_nonterm4

n'en sont pas.

A.1.3 *Règles de production*

Les règles de production relatives aux langages littéraux de programmation d'automates programmables doivent former une *grammaire étendue* dans laquelle chaque règle a la forme suivante:

symbole non_terminal ::= structure_étendue

Cette règle peut se lire de la façon suivante:

"Un symbole non_terminal peut se composer d'une structure_étendue."

Les structures étendues peuvent être construites en respectant les règles suivantes:

1) Le cordon nul, NIL, est une structure étendue;
2) Un symbole terminal est une structure étendue;
3) Un symbole non terminal est une structure étendue;
4) Si S est une structure étendue, les expressions suivantes sont également des structures étendues:

   (S), représentant S proprement dit;

   {S}, *fermeture,* représentant un certain nombre (éventuellement zéro) d'enchaînements de S;

   [S], *option,* représentant zéro ou une occurrence de S.

5) si S1 et S2 sont des structures étendues, les expressions suivantes sont également des structures étendues:

   S1| S2, *alternance,* représentant un choix entre S1 et S2;

   S1 S2, *enchaînement,* signifiant que S1 doit être suivi de S2.

6) l'enchaînement *précède* l'alternance, c'est-à-dire que S1 | S2 S3 est équivalent à S1 | (S2 S3), et S1 S2 | S3 est équivalent à (S1 S2) | S3.

A.1.2 *Non-terminal symbols*

Non-terminal textual symbols shall be represented by strings of lower-case letters, numbers, and the underline character (_), beginning with a lower-case letter. For instance, the strings

nonterm1

and

non_term_2

are valid non-terminal symbols, while the strings

3nonterm

and

_nonterm4

are not.

A.1.3 *Production rules*

The production rules for textual programmable controller programming languages shall form an *extended grammar* in which each rule has the form

non_terminal_symbol ::= extended_structure

This rule can be read as:

"A non_terminal_symbol can consist of an extended_structure."

Extended structures can be constructed according to the following rules:

1) The null string, NIL, is an extended structure.
2) A terminal symbol is an extended structure.
3) A non-terminal symbol is an extended structure.
4) If S is an extended structure, then the following expressions are also extended structures:

   (S), meaning S itself.

   {S}, *closure*, meaning zero or more concatenations of S.

   [S], *option*, meaning zero or one occurrence of S.
5) If S1 and S2 are extended structures, then the following expressions are extended structures:

   S1| S2, *alternation*, meaning a choice of S1 or S2.

   S1 S2, *concatenation*, meaning S1 followed by S2.
6) Concatenation *precedes* alternation, that is, S1 | S2 S3 is equivalent to S1 | (S2 S3), and S1 S2 | S3 is equivalent to (S1 S2) | S3.

A.2 Sémantique

La sémantique des langages de programmation littéraux d'automates programmables est définie dans la présente partie par des textes en clair accompagnant les règles de production et faisant référence aux descriptions fournies dans les paragraphes appropriés. Les options standards offertes à l'utilisateur et au fabricant sont spécifiées dans cette sémantique.

Dans certains cas, il est plus pratique d'intégrer les informations sémantiques dans une structure étendue. Dans ces cas, ces informations sont délimitées par une paire de signes < >, par exemple, <information sémantique>.

A.2 Semantics

Programmable controller textual programming language semantics are defined in this part by appropriate natural language text, accompanying the production rules, which references the descriptions provided in the appropriate clauses. Standard options available to the user and manufacturer are specified in these semantics.

In some cases it is more convenient to embed semantic information in an extended structure. In such cases, this information is delimited by paired angle brackets, for example, <semantic information>.

Annexe B
(normative)

Spécifications formelles des éléments de langage

B.0 Modèle de programmation

Cette annexe et normative dans le sens qu'un compilateur capable de reconnaître la totalité de la syntaxe de cette annexe doit être capable de reconnaître la syntaxe de toute réalisation en langage littéral conforme à cette norme.

RÈGLES DE PRODUCTION:

nom_de_l'élément_de_bibliothèque ::= nom_du_type_de_donnée | nom_de_fonction
| nom_du_type_de_bloc_fonctionnel | nom_du_type_de_programme
| nom_du_type_de_ressource | nom_de_la_configuration

déclaration_de_l'élément_de_bibliothèque ::= déclaration_du_type_de_donnée
| déclaration_de_fonction | déclaration_du_bloc_fonctionnel
| déclaration_du_programme | déclaration_de_configuration

SÉMANTIQUE: Ces productions reflètent le modèle de programmation type, défini en 1.4.3, dans lequel les *déclarations* représentent le mécanisme de base nécessaire à la production d'*éléments de bibliothèque* nommés. La syntaxe et la sémantique des symboles non terminaux donnés ci-dessus sont définies dans les paragraphes énumérés ci-après.

Symbole non terminal	Syntaxe	Sémantique
nom_du_type_de_donnée déclaration_du_type_de_donnée	B.1.3	2.3
nom_de_fonction déclaration_de_fonction	B.1.5.1	2.5.1
nom_du_type_de_bloc_fonctionnel déclaration_du_bloc_fonctionnel	B.1.5.2	2.5.2
nom_du_type_de_programme déclaration_du_programme	B.1.5.3	2.5.3
nom_du_type_de_ressource nom_de_la_configuration déclaration_de_la_configuration	B.1.7	2.7

B.1 Eléments communs

B.1.1 *Lettres, chiffres et identificateurs*

RÈGLES DE PRODUCTION:

lettre ::= 'A'|'B'|<...>|'Z'|'a'|'b'|<...>|'z'

chiffre ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

chiffre_octal ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'

chiffre_hexa ::= chiffre|'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f'

identificateur ::= (lettre|('_'(lettre|chiffre))) {['_'](lettre|chiffre)}

Annex B
(normative)

Formal specifications of language elements

B.0 Programming model

The contents of this annex are normative in the sense that a compiler which is capable of recognizing all the syntax in this annex shall be capable of recognizing the syntax of any textual language implementation complying with this standard.

PRODUCTION RULES:

library_element_name ::= data_type_name | function_name
| function_block_type_name | program_type_name
| resource_type_name | configuration_name

library_element_declaration ::= data_type_declaration
| function_declaration | function_block_declaration
| program_declaration | configuration_declaration

SEMANTICS: These productions reflect the basic programming model defined in 1.4.3, where *declarations* are the basic mechanism for the production of named *library elements.* The syntax and semantics of the non-terminal symbols given above are defined in the subclauses listed below.

Non-terminal symbol	Syntax	Semantic
data_type_name data_type_declaration	B.1.3	2.3
function_name function_declaration	B.1.5.1	2.5.1
function_block_type_name function_block_declaration	B.1.5.2	2.5.2
program_type_name program_declaration	B.1.5.3	2.5.3
resource_type_name configuration_name configuration_declaration	B.1.7	2.7

B.1 Common elements

B.1.1 *Letters, digits and identifiers*

PRODUCTION RULES:

letter ::= 'A'|'B'|<...>|'Z'|'a'|'b'|<...>|'z'

digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

octal_digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'

hex_digit ::= digit|'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f'

identifier ::= (letter|('_'(letter|digit))) {['_'](letter|digit)}

SÉMANTIQUE:

L'ellipse <...> représente ici la suite normale des 26 lettres de l'ISO/IEC 646.

Il est possible d'utiliser les caractères appartenant aux jeux de caractères nationaux; cependant, la portabilité internationale de la représentation imprimée des programmes ne peut, dans ce cas, être garantie.

Le format (majuscule ou minuscule) des lettres doit être significatif dans les symboles terminaux, mais non dans les autres éléments syntaxiques.

B.1.2 *Constantes*

RÈGLE DE PRODUCTION:

constante ::= libellé_numérique|cordon_de_caractères|libellé_temporel

SÉMANTIQUE:

Les représentations externes des données décrites en 2.2 sont désignées comme "constantes" dans la présente annexe.

B.1.2.1 *Libellés numériques*

RÈGLES DE PRODUCTION:

libellé_numérique ::= libellé_entier | libellé_réel
libellé_entiee ::= entier_signé | entier_binaire | entier_octal | entier_hexa
entier_signé ::= ['+'|'-'] entier
entier ::= chiffre {['_'] chiffre}
entier_binaire ::= '2#' bit {['_'] bit}
bit ::= '1'|'0'
entier_octal ::= '8#' chiffre_octal {['_'] chiffre_octal}
entier_hexa ::= '16#' chiffre_hexa {['_'] chiffre_hexa}
libellé_réel ::= entier_signé '.' entier [exposant]
exposant ::= ('E'|'e') ['+'|'-'] entier

SÉMANTIQUE: Voir 2.2.1.

B.1.2.2 *Cordons de caractères*

RÈGLES DE PRODUCTION:

cordon_de_caractères ::= " ' " {représentation_du_caractère} " ' "
représentation_du_caractère ::= <tout caractère imprimable sauf '$'>
|'$' chiffre_hexa chiffre_hexa |'$$'
|"$'"|'$L'|'$N'|'$P'|'$R'|'$T'|'$l'|'$n'|'$p'|'$r'|'$t'

SÉMANTIQUE: Voir 2.2.2.

SEMANTICS:

The ellipsis <...> here indicates the ISO/IEC 646 sequence of 26 letters.

Characters from national character sets can be used; however, international portability of the printed representation of programs cannot be guaranteed in this case.

The case of letters shall be significant in terminal symbols, but not in other syntactic elements.

B.1.2 *Constants*

PRODUCTION RULE:

constant ::= numeric_literal | character_string | time_literal

SEMANTICS:

The external representations of data described in 2.2 are designated as "constants" in this annex.

B.1.2.1 *Numeric literals*

PRODUCTION RULES:

numeric_literal ::= integer_literal | real_literal

integer_literal ::= signed_integer | binary_integer | octal_integer | hex_integer

signed_integer ::= ['+' |'-'] integer

integer ::= digit {['_'] digit}

binary_integer ::= '2#' bit {['_'] bit}

bit ::= '1' | '0'

octal_integer ::= '8#' octal_digit {['_'] octal_digit}

hex_integer ::= '16#' hex_digit {['_'] hex_digit}

real_literal ::= signed_integer '.' integer [exponent]

exponent ::= ('E' | 'e') ['+'|'-'] integer

SEMANTICS: See 2.2.1.

B.1.2.2 *Character strings*

PRODUCTION RULES:

character_string ::= " ' " {character_representation} " ' "

character_representation ::= <any printable character except '$'> |'$' hex_digit hex_digit |'$$' |"$'"|'$L'|'$N'|'$P'|'$R'|'$T'|'$l'|'$n'|'$p'|'$r'|'$t'

SEMANTICS: See 2.2.2.

B.1.2.3 *Libellés temporels*

RÈGLE DE PRODUCTION:

libellé_temporel ::= durée | heure_du_jour | date | date_et_heure

SÉMANTIQUE: Voir 2.2.3.

B.1.2.3.1 *Durée*

RÈGLES DE PRODUCTION:

durée ::= ('T'|'t'|'TIME'|'time') '#' ['-'] intervalle
intervalle ::= jours | heures | minutes | secondes | millisecondes
jours ::= ('d'|'D') virgule_fixe | entier ('d'|'D') ['_'] heures
virgule_fixe ::= entier ['.' entier]
heures ::= ('h'|'H') virgule_fixe | entier ('h'|'H') ['_'] minutes
minutes ::= ('m'|'M') virgule_fixe | entier ('m'|'M') ['_'] secondes
secondes ::= ('s'|'S') virgule_fixe | entier ('s'|'S') ['_'] millisecondes
millisecondes ::= ('ms'|'MS') virgule_fixe

SÉMANTIQUE: Voir 2.2.3.1.

NOTE - La sémantique de 2.2.3.1 impose des contraintes supplémentaires aux valeurs autorisées pour les heures, les minutes, les secondes et les millisecondes.

B.1.2.3.2 *Heure du jour et date*

RÈGLES DE PRODUCTION

heure_du_jour ::= ('TIME_OF_DAY'|'time_of_day'|'TOD'|'tod') '#' heure_du_jour
heure_du_jour ::= heure_du_jour ':' minute_du_jour ':' seconde_du_jour
heure_du_jour ::= entier
minute_du_jour ::= entier
seconde_du_jour ::= virgule_fixe
date ::= ('DATE'|'date'|'D'|'d') '#' libellé_de_date
libellé_de_date ::= année '-' mois '-' jour
année ::= entier
mois ::= entier
jour ::= entier
date_et_heure ::= ('DATE_AND_TIME'|'date_and_time'|'DT' |'dt') '#' libellé_de_date '-' heure_du_jour

SÉMANTIQUE: Voir 2.2.3.2.

NOTE - La sémantique de 2.2.3.2 impose des contraintes supplémentaires aux valeurs autorisées pour heure_du_jour, minute_du_jour, seconde_du_jour, année, mois et jour.

B.1.2.3 *Time literals*

PRODUCTION RULE:

time_literal ::= duration | time_of_day | date | date_and_time

SEMANTICS: See 2.2.3.

B.1.2.3.1 *Duration*

PRODUCTION RULES:

duration ::= ('T'|'t'|'TIME'|'time') '#' ['-'] interval
interval ::= days | hours | minutes | seconds | milliseconds
days ::= fixed_point ('d'|'D') | integer ('d'|'D') ['_'] hours
fixed_point ::= integer ['.' integer]
hours ::= fixed_point ('h'|'H') | integer ('h'|'H') ['_'] minutes
minutes ::= fixed_point ('m'|'M') | integer ('m'|'M') ['_'] seconds
seconds ::= fixed_point ('s'|'S') | integer ('s'|'S') ['_'] milliseconds
milliseconds ::= fixed_point ('ms'|'MS')

SEMANTICS: See 2.2.3.1.

NOTE - The semantics of 2.2.3.1 impose additional constraints on the allowable values of hours, minutes, seconds, and milliseconds.

B.1.2.3.2 *Time of day and date*

PRODUCTION RULES:

time_of_day ::= ('TIME_OF_DAY'|'time_of_day'|'TOD'|'tod') '#' daytime
daytime ::= day_hour ':' day_minute ':' day_second
day_hour ::= integer
day_minute ::= integer
day_second ::= fixed_point
date ::= ('DATE'|'date'|'D'|'d') '#' date_literal
date_literal ::= year '-' month '-' day
year ::= integer
month ::= integer
day ::= integer
date_and_time ::= ('DATE_AND_TIME'|'date_and_time'|'DT'|'dt') '#' date_literal '-' daytime

SEMANTICS: See 2.2.3.2.

NOTE - The semantics of 2.2.3.2 impose additional constraints on the allowable values of day_hour, day_minute, day_second, year, month, and day.

B.1.3 *Types de données*

RÈGLES DE PRODUCTION:

nom_du_type_de_donnée ::= nom_du_type_non_générique|nom_du_type_générique

nom_du_type_non_générique ::= nom_du_type_élémentaire | nom_du_type_dérivé

SÉMANTIQUE: Voir 2.3.

B.1.3.1 *Types de données élémentaires*

RÈGLES DE PRODUCTION:

nom_du_type_élémentaire ::= nom_du_type_numérique|nom_du_type_date
| nom_du_type_de_cordon_de_bits |'STRING'|'TIME'

nom_du_type_numérique ::= nom_du_type_entier | nom_du_type_réel

nom_du_type_entier ::= nom_du_type_entier_signé | nom_du_type_entier_non_signé

nom_du_type_entier_signé ::= 'SINT'|'INT'|'DINT'|'LINT'

nom_du_type_entier_non_signé ::= 'USINT'|'UINT'|'UDINT' |'ULINT'

nom_du_type_réel ::= 'REAL'|'LREAL'

nom_du_type_date ::= 'DATE'|'TIME_OF_DAY'|'TOD' |'DATE_AND_TIME'|'DT'

nom_du_type_de_cordon_de_bits ::= 'BOOL'|'BYTE'|'WORD'|'DWORD' |'LWORD'

SÉMANTIQUE: Voir 2.3.1.

B.1.3.2 *Types de données génériques*

RÈGLE DE PRODUCTION:

nom_du_type_générique ::= 'ANY'|'ANY_NUM'|'ANY_REAL'|'ANY_INT'|'ANY_BIT'
|'ANY_DATE'

SÉMANTIQUE : Voir 2.3.2.

B.1.3.3 Types de données dérivées

RÈGLES DE PRODUCTION:

nom_du_type_dériv :: =nom_du_type_élément_simple | nom_du_type_tableau
| nom_du_type_structure

nom_du_type_élément_simple ::= nom_du_type_simple
| nom_du_type_intervalle | nom_du_type_énuméré

nom_du_type_simple ::= identificateur

nom_du_type_intervalle ::= identificateur

nom_du_type_énuméré ::= identificateur

nom_du_type_tableau ::= identificateur

nom_du_type_structure ::= identificateur

déclaration_de_type_de_donnée ::= 'TYPE' déclaration de type ';'
{déclaration_de_type ';'} 'END_TYPE'

B.1.3 *Data types*

PRODUCTION RULES:

data_type_name ::= non_generic_type_name | generic_type_name

non_generic_type_name ::= elementary_type_name | derived_type_name

SEMANTICS: See 2.3.

B.1.3.1 *Elementary data types*

PRODUCTION RULES:

elementary_type_name ::= numeric_type_name | date_type_name
| bit_string_type_name |'STRING'|'TIME'

numeric_type_name ::= integer_type_name | real_type_name

integer_type_name ::= signed_integer_type_name | unsigned_integer_type_name

signed_integer_type_name ::= 'SINT'|'INT'|'DINT'|'LINT'

unsigned_integer_type_name ::= 'USINT'|'UINT'|'UDINT'|'ULINT'

real_type_name ::= 'REAL'|'LREAL'

date_type_name ::= 'DATE'|'TIME_OF_DAY'|'TOD'|'DATE_AND_TIME'|'DT'

bit_string_type_name ::= 'BOOL'|'BYTE'|'WORD'|'DWORD'|'LWORD'

SEMANTICS: See 2.3.1.

B.1.3.2 *Generic data types*

PRODUCTION RULE:

generic_type_name ::= 'ANY'|'ANY_NUM'|'ANY_REAL'|'ANY_INT'|'ANY_BIT'
|'ANY_DATE'

SEMANTICS: See 2.3.2.

B.1.3.3 *Derived data types*

PRODUCTION RULES:

derived_type_name ::= single_element_type_name | array_type_name
| structure_type_name

single_element_type_name ::= simple_type_name | subrange_type_name
| enumerated_type_name

simple_type_name ::= identifier

subrange_type_name ::= identifier

enumerated_type_name ::= identifier

array_type_name ::= identifier

structure_type_name ::= identifier

data_type_declaration ::= 'TYPE' type_declaration ';' {type_declaration ';'} 'END_TYPE'

déclaration_de_type ::= déclaration_de_type_élément_simple
| déclaration_de_type_tableau | déclaration_de_type_structure

déclaration_de_type_élément_simple ::= déclaration_de_type_simple
| déclaration_de_type_intervalle | déclaration_de_type_énuméré

déclaration_de_type_simple ::= nom_du_type_simple ':' init_spéc_simple

init_spéc_simple ::= spécification_simple [':=' constante]

spécification_simple ::= nom_du_type_élémentaire | nom_du_type_simple

déclaration_de_type_intervalle ::= nom_du_type_intervalle ':' init_spéc_d'intervalle

init_spéc_d'intervalle ::= spécification_d'intervalle [':=' entier_signé]

spécification_d'intervalle ::= nom_du_type_entier '(' intervalle ')'
| nom_du_type_d'intervalle

intervalle ::= entier_signé '..' entier_signé

déclaration_de_type_énuméré ::= nom_du_type_énuméré ':' init_spéc_énumérée

init_spéc_énumérée ::= spécification_énumérée [':=' valeur_énumérée]

spécification_énumérée ::= ('(' valeur_énumérée {',' valeur_énumérée} ')')
| nom_du_type_énuméré

valeur_énumérée ::= identificateur

déclaration_du_type_tableau ::= nom_du_type_tableau ':' init_spéc_tableau

init_spéc_tableau ::= spécification_de_tableau [':=' initialisation_de_tableau]

spécification_de_tableau ::= nom_du_type_tableau
|'ARRAY' '[' intervalle (',' intervalle) ']' 'OF' nom_du_type_non_générique

initialisation_de_tableau ::= éléments_initiaux_de_tableau
{',' éléments_initiaux_de_tableau}

éléments_initiaux_de_tableau ::= élément_initial_de_tableau
| entier '(' élément_initial_de_tableau ')'

élément_initial_de_tableau ::= constante | valeur_énumérée
| initialisation_de_structure | initialisation_de_tableau

déclaration_de_type_structure ::= nom_du_type_structure ':' spécification_de_structure

spécification_de_structure ::= déclaration_de_structure | structure_initialisée

structure_initialisée ::= nom_du_type_de_structure [initialisation_de_structure]

déclaration_de_structure ::= 'STRUCT' déclaration_d'élément_de_structure ';'
{déclaration_d'élément_de_structure ';'} 'END_STRUCT'

déclaration_d'élément_de_structure ::= nom_d'élément_de_structure ':'
(init_spéc_simple | init_spéc_intervalle | init_spéc_énumérée | init_spéc_tableau
| structure_initialisée)

nom_d'élément_de_structure ::= identificateur

initialisation_de_structure ::= '(' initialisation_d'élément_de_structure
{',' initialisation_d'élément_de_structure} ')'

initialisation_d'élément_de_structure ::= nom_d'élément_de structure ':='
(constante | valeur_énumérée | initialisation_de_tableau | initialisation_de_structure)

SÉMANTIQUE: Voir 2.3.3.

type_declaration ::= single_element_type_declaration | array_type_declaration
| structure_type_declaration

single_element_type_declaration ::= simple_type_declaration
| subrange_type_declaration | enumerated_type_declaration

simple_type_declaration ::= simple_type_name ':' simple_spec_init

simple_spec_init ::= simple_specification [':=' constant]

simple_specification ::= elementary_type_name | simple_type_name

subrange_type_declaration ::= subrange_type_name ':' subrange_spec_init

subrange_spec_init ::= subrange_specification [':=' signed_integer]

subrange_specification ::= integer_type_name '(' subrange')' | subrange_type_name

subrange ::= signed_integer '..' signed_integer

enumerated_type_declaration ::= enumerated_type_name ':' enumerated_spec_init

enumerated_spec_init ::= enumerated_specification [':=' enumerated_value]

enumerated_specification ::= ('(' enumerated_value {',' enumerated_value} ')')
| enumerated_type_name

enumerated_value ::= identifier

array_type_declaration ::= array_type_name ':' array_spec_init

array_spec_init ::= array_specification [':=' array_initialization]

array_specification ::= array_type_name
| 'ARRAY' '[' subrange {',' subrange} ']' 'OF' non_generic_type_name

array_initialization ::= array_initial_elements {',' array_initial_elements}

array_initial_elements ::= array_initial_element | integer '(' array_initial_element ')'

array_initial_element ::= constant | enumerated_value | structure_initialization
| array_initialization

structure_type_declaration ::= structure_type_name ':' structure_specification

structure_specification ::= structure_declaration | initialized_structure

initialized_structure ::= structure_type_name [structure_initialization]

structure_declaration ::= 'STRUCT' structure_element_declaration ';'
{struture_element_declaration ';'} 'END_STRUCT'

structure_element_declaration ::= structure_element_name ':'
(simple_spec_init | subrange_spec_init | enumerated_spec_init | array_spec_init
| initialized_structure)

structure_element_name ::= identifier

structure_initialization ::= '(' structure_element_initialization
{',' structure_element_initialization} ')'

structure_element_initialization ::= structure_element_name ':='
(constant | enumerated_value | array_initialization | structure_initialization)

SEMANTICS: See 2.3.3.

B.1.4 *Variables*

RÈGLES DE PRODUCTION:

variable ::= variable_directe | variable_symbolique

variable_symbolique ::= nom_de_variable | variable_multi-élément

nom_de_variable ::= identificateur

SÉMANTIQUE: Voir 2.4.1.

B.1.4.1 *Variables Représentées Directement*

RÈGLES DE PRODUCTION:

variable_directe ::= '%' préfixe_d'emplacement préfixe_de_taille entier {'.' entier}

préfixe_d'emplacement ::= 'I'|'Q'|'M'

préfixe_de_taille ::= NIL| 'X' | 'B' | 'W' | 'D' | 'L'

SÉMANTIQUE: Voir 2.4.1.1.

B.1.4.2 *Variables multi-éléments*

RÈGLES DE PRODUCTION:

variable_multi-éléments ::= variable_tableau | variable_structurée

variable_tableau ::= variable_indice liste_d'indices

variable_indice ::= variable_symbolique

liste_d'indices ::= '[' indice {',' indice} ']'

indice ::= expression

variable_structurée ::= variable_d'enregistrement '.' sélecteur_de_champ

variable_d'enregistrement ::= variable_symbolique

sélecteur_de_champ ::= identificateur

SÉMANTIQUE: Voir 2.4.1.2.

B.1.4.3 *Déclaration et initialisation*

RÈGLES DE PRODUCTION:

input_declarations ::= 'VAR_INPUT' déclaration_d'entrée ';' {input_declaration ';'} 'END_VAR'

input_declaration ::= var_init_decl | edge_declaration

edge_declaration ::= var1_list ':' 'BOOL' ['R_EDGE'|'F_EDGE']

var_init_decl ::= varl_init_decl | array_var_init_decl | structured_var_init_decl | fb_name_decl

var1_init_decl ::= var1_list ':' (simple_spec_init | subrange_spec_init | enumerated_spec_init)

var1_list ::= variable_name {',' variable_name}

array_var_init_decl ::= varl_list ':' array_spec_init

B.1.4 *Variables*

PRODUCTION RULES:

variable ::= direct_variable | symbolic_variable

symbolic_variable ::= variable_name | multi_element_variable

variable_name ::= identifier

SEMANTICS: See 2.4.1.

B.1.4.1 *Directly represented variables*

PRODUCTION RULES:

direct_variable ::= '%' location_prefix size_prefix integer {'.' integer}

location_prefix ::= 'I'|'Q'|'M'

size_prefix ::= NIL| 'X' | 'B' | 'W' | 'D' | 'L

SEMANTICS: See 2.4.1.1.

B.1.4.2 *Multi-element variables*

PRODUCTION RULES:

multi_element_variable ::= array_variable | structured_variable

array_variable ::= subscripted_variable subscript_list

subscripted_variable ::= symbolic_variable

subscript_list ::= '[' subscript {',' subscript} ']'

subscript ::= expression

structured_variable ::= record_variable '.' field_selector

record_variable ::= symbolic_variable

field_selector ::= identifier

SEMANTICS: See 2.4.1.2.

B.1.4.3 *Declaration and initialization*

PRODUCTION RULES:

input_declarations ::= 'VAR_INPUT' input_declaration ';' {input_declaration ';'} 'END_VAR'

input_declaration ::= var_init_decl | edge_declaration

edge_declaration ::= var1_list ':' 'BOOL' ['R_EDGE'|'F_EDGE']

var_init_decl ::= var1_init_decl | array_var_init_decl | structured_var_init_decl | fb_name_decl

var1_init_decl ::= var1_list ':' (simple_spec_init | subrange_spec_init | enumerated_spec_init)

var1_list ::= variable_name {',' variable_name}

array_var_init_decl ::= var1_list ':' array_spec_init

structured_var_init_decl ::= var1_list ':' initialized_structure

fb_name_decl ::= fb_name_list ':' nom_du_type_de_bloc_fonctionnel

fb_name_list ::= fb_name {',' fb_name}

fb_name ::= identifier

output declarations ::= 'VAR_OUTPUT' ['RETAIN'] var_init_decl ';' {var_init_decl ';'}
'END_VAR'

input_output_declarations ::= 'VAR_IN_OUT' var_declaration ';' {var_declaration ';'}
'END_VAR'

var_declaration ::= var1_declaration | array_var_declaration
| structured_var_declaration | fb_name_decl

var1_declaration ::= var1_list ':' (simple_specification|
subrange_specification | enumerated_specification)

array_var_declaration ::= var1_list ':' array_specification

structured_var_declaration ::= var1_list ':' structure_type_name

var_declarations ::= 'VAR' ['CONSTANT'] var_init_decl ';' {var_init_decl ';'}
'END_VAR'

retentive_var_declarations ::= 'VAR' 'RETAIN' var_init_decl ';' {var_init_decl ';'}
'END_VAR'

located_var_declarations ::= 'VAR' ['CONSTANT'] ['RETAIN'] located_var_decl ';'
{located_var_decl ';'} 'END_VAR'

located_var_decl ::= [variable_name] location ':' located_var_spec_init

external_var_declarations ::= 'VAR_EXTERNAL' external_declaration ';'
{external_declaration ';'} 'END_VAR'

external_declaration ::= global_var_name ':' (simple_specification | subrange_
specification | | enumerated_specification
| array_specification | structure_type_name | function_block_type_name)

global_var_name ::= identifier

global_var_declarations ::= 'VAR_GLOBAL' ['CONSTANT'] ['RETAIN']
global_var_decl ';' {global_var_decl ';'} 'END_VAR'

global_var_decl ::= global_var_spec ':' located_var_spec_init

Global_var_spec ::= global_var_list | [global_var_name] location

located_var_spec_init ::= simple_spec_init | subrange_spec_init | enumerated_spec_init
| array_spec_init | initialized_structure

location ::= 'AT' direct_variable

global_var_list ::= global_var_name {',' global_var_name}

SÉMANTIQUE: Voir 2.4.2. Le non-terminal "function_block_type_name" est défini en B.1.5.2.

B.1.5 *Unités d'organisation de programme*

B.1.5.1 *Fonctions*

RÈGLES DE PRODUCTION:

nom_de_fonction ::= nom_de_fonction_standard | nom_de_fonction_dérivée

nom_de_fonction_standard ::= <tel que défini en 2.5.1.5>

structured_var_init_decl ::= var1_list ':' initialized_structure

fb_name_decl ::= fb_name_list ':' function_block_type_name

fb_name_list ::= fb_name {',' fb_name}

fb_name ::= identifier

output_declarations ::= 'VAR_OUTPUT' ['RETAIN'] var_init_decl ';' {var_init_decl ';'}
'END_VAR'

input_output_declarations ::= 'VAR_IN_OUT' var_declaration ';' {var_declaration ';'}
'END_VAR'

var_declaration ::= var1_declaration | array_var_declaration
| structured_var_declaration | fb_name_decl

var1_declaration ::= var1_list ':' (simple_specification
| subrange_specification | enumerated_specification)

array_var_declaration ::= var1_list ':' array_specification

structured_var_declaration ::= var1_list ':' structure_type_name

var_declarations ::= 'VAR' ['CONSTANT'] var_init_decl ';' {(var_init_decl ';') }
'END_VAR'

retentive_var_declarations ::= 'VAR' 'RETAIN' var_init_decl ';' {var_init_decl ';'}
'END_VAR'

located_var_declarations ::= 'VAR' ['CONSTANT'] ['RETAIN'] located_var_decl ';'
{located_var_decl ';'} 'END_VAR'

located_var_decl ::= [variable_name] location ':' located_var_spec_init

external_var_declarations ::= 'VAR_EXTERNAL' external_declaration ';'
{external_declaration ';'} 'END_VAR'

external_declaration ::= global_var_name ':' (simple_specification | subrange_
specification | enumerated_specification
| array_specification | structure_type_name | function_block_type_name)

global_var_name ::= identifier

global_var_declarations ::= 'VAR_GLOBAL' ['CONSTANT'] ['RETAIN']
global_var_decl ':' (global_var_decl ';'} 'END_VAR'

global_var_decl ::= global_var_spec ':' located_var_spec_init

global_var_spec ::= global_var_list | [global_var_name] location

located_var_spec_init ::= simple_spec_init | subrange_spec_init | enumerated_spec_init
| array_spec_init | initialized_structure

location ::= 'AT' direct_variable

global_var_list ::= global_var_name {',' global_var_name}

SEMANTICS: See 2.4.2. The non-terminal "function_block_type_name" is defined in B.1.5.2.

B.1.5 *Program organization units*

B.1.5.1 *Functions*

PRODUCTION RULES:

function_name ::= standard_function_name | derived_function_name

standard_function_name ::= <as defined in 2.5.1.5>

nom_de_fonction_dérivée ::= identificateur

déclaration de fonction ::=
'FUNCTION' derived_function_name ':' (elementary_type_name | derived_type_name)
input_declarations
['VAR' [CONSTANT'] function_var_decls 'END_VAR']
function_body
'END_FUNCTION'

function_var_decls ::= function_var_decl ';' {function_var_decl ';'}

function_var_decl ::= varl_declaration | array_var_declaration
| structured_var_declaration

function_body ::= ladder_diagram | function_block_diagram | instruction_list
| statement_list

SÉMANTIQUE: Voir 2.5.1.

NOTES

1 Cette syntaxe ne prend pas en compte le fait que les références de bloc fonctionnel et les invocations ne sont pas permises dans les corps de fonction.

2 Les diagrammes à contacts et les diagrammes à bloc fonctionnel sont représentés graphiquement comme cela est défini à l'article 4. La liste_d'_instruction et la liste d'énoncé sont définies respectivement en B.2.1 et B.3.2.

B.1.5.2 *Blocs fonctionnels*

RÈGLES DE PRODUCTION:

nom_de_type_de_bloc_fonctionnel ::= nom_de_bloc_fonctionnel_standard
| nom_de_bloc_fonctionnel_dérivé

nom_de_bloc_fonctionnel_standard ::= <tel que défini en 2.5.2.3«>

nom_de_bloc_fonctionnel_dérivé ::= identificateur

déclaration_de_bloc_fonctionnel ::=
'FUNCTION_BLOCK' derived_function_block_name
{fb_io_var_declarations}
{other_var_declarations}
function_block_body
'END_FUNCTION_BLOCK'

fb_io_var_declarations ::= input_declarations | output_declarations
| input_output_declarations

other_var_declarations ::= external_var_declarations | var_declarations
| retentive_var_declarations

function_block_body ::= sequential_function_chart | ladder_diagram
| function_block_diagram | instruction_list | statement_list

SÉMANTIQUE: Voir 2.5.2.

NOTES

1 Les diagrammes à contacts et les diagrammes à bloc fonctionnel sont représentés graphiquement comme cela est défini à l'article 4.

2 Le schéma_de_fonction_séquentielle, la liste_d'_instruction et la liste_d'_énoncé sont définis respectivement en B.1.6, B.2 et B.3.2.

derived_function_name ::= identifier

function_declaration ::=
'FUNCTION' derived_function_name ':' (elementary_type_name | derived_type_name)
input_declarations
['VAR' ['CONSTANT'] function_var_decls 'END_VAR']
function_body
'END_FUNCTION'

function_var_decls ::= function_var_decl ';' {function_var_decl ';'}

function_var_decl ::= var1_declaration | array_var_declaration
| structured_var_declaration

function_body ::= ladder_diagram | function_block_diagram | instruction_list
| statement_list

SEMANTICS: See 2.5.1.

NOTES

1 This syntax does not reflect the fact that function block references and invocations are not allowed in function bodies.

2 Ladder diagrams and function block diagrams are graphically represented as defined in clause 4. The non-terminals *instruction_list* and *statement_list* are defined in B.2.1 and B.3.2, respectively.

B.1.5.2 *Function blocks*

PRODUCTION RULES:

function_block_type_name ::= standard_function_block_name | derived_function_block_name

standard_function_block_name ::= <as defined in 2.5.2.3>

derived_function_block_name ::= identifier

function_block_declaration ::=
'FUNCTION_BLOCK' derived_function_block_name
{fb_io_var_declarations}
{other_var_declarations}
function_block_body
'END_FUNCTION_BLOCK'

fb_io_var_declarations ::= input_declarations | output_declarations
| input_output_declarations

other_var_declarations ::= external_var_declarations | var_declarations
| retentive_var_declarations

function_block_body ::= sequential_function_chart | ladder_diagram
| function_block_diagram | instruction_list | statement_list

SEMANTICS: See 2.5.2.

NOTES

1 Ladder diagrams and function block diagrams are graphically represented as defined in clause 4.

2 The non-terminals *sequential_function_chart, instruction_list,* and *statement_list* are defined in B.1.6, B.2, and B.3.2, respectively.

B.1.5.3 *Programmes*

RÈGLES DE PRODUCTION:

nom_de_type_de_programme ::= identificateur

déclaration_de_programme ::=
'PROGRAM' program_type_name
{fb_io_var_declarations}
{other_var_declarations | located_var_declarations}
[program_access_decls]
function_block_body
'END_PROGRAM'

program_access_decls ::=
'VAR_ACCESS' program_access_decl ';'
{program_access_decl ';'}
'END_VAR'

program_access_decl ::= access_name ':' symbolic_variable ':'
non_generic_type_name direction

SÉMANTIQUE: Voir 2.5.3.

B.1.6 *Eléments de diagramme fonctionnel en séquence*

RÈGLES DE PRODUCTION:

diagramme fonctionnel en séquence ::= réseau_diagramme fonctionnel en séquence
{réseau_diagramme fonctionnel en séquence}

réseau_diagramme fonctionnel en séquence ::= étape_initiale {étape | transition | action}

étape_initiale ::= 'INITIAL_STEP' nom_de_l'étape ':' {association_d'actions ';'} 'END_STEP'

étape ::= 'STEP' nom_de_l'étape ':' {association_d'actions ';'} 'END_STEP'

nom_de_l'étape ::= identificateur

association_d'actions ::= nom_d'action '(' qualificatif_d'action {',' nom_d'asservissement} ')'

nom_d'action ::= identificateur

qualificatif_d'action ::= 'N' | 'R' | 'S' | 'P' | qualificatif_temporel ',' temps_d'action

qualificatif_temporel ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'

temps_d'action ::= durée | nom_de_variable

nom_d'asservissement ::= nom_de_variable

transition ::= 'TRANSITION' (transition_name | unnamed_trans) transition_condition
'END_TRANSITION'

unnamed_trans ::= 'FROM' steps 'TO' steps

nom_de_transition ::= identificateur

étapes ::= nom_de_l'étape | '(' nom_de_l'étape ',' nom_de_l'étape {',' nom_de_l'étape} ')'

condition_de_transition ::= ':' liste_d'instructions |':=' expression ';' | ':' (réseau_fbd | échelon)

action ::= 'ACTION' nom_d'action ':'
corps_de_bloc_fonctionnel
'END_ACTION'

B.1.5.3 *Programs*

PRODUCTION RULES:

```
program_type_name ::= identifier

program_declaration ::=
      'PROGRAM' program_type_name
        {fb_io_var_declarations}
        {other_var_declarations | located_var_declarations | global_var_declarations}
        [program_access_decls]
        function_block_body
      'END_PROGRAM'

program_access_decls ::=
      'VAR_ACCESS' program_access_decl ';'
        {program_access_decl ';' }
      'END_VAR'

program_access_decl ::= access_name ':' symbolic_variable ':'
          non_generic_type_name direction
```

SEMANTICS: See 2.5.3.

B.1.6 *Sequential function chart elements*

PRODUCTION RULES:

```
sequential_function_chart ::= sfc_network {sfc_network}

sfc_network ::= initial_step {step | transition | action}
initial_step ::= 'INITIAL_STEP' step_name ':' {action_association ';'} 'END_STEP'
step ::= 'STEP' step_name ':' {action_association ';'} 'END_STEP'
step_name ::= identifier
action_association ::= action_name '(' action_qualifier {',' feedback_name} ')'
action_name ::= identifier
action_qualifier ::= 'N' | 'R' | 'S' | 'P' | timed_qualifier ',' action_time
timed_qualifier ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'
action_time ::= duration | variable_name
feedback_name ::= variable_name
transition ::= 'TRANSITION' (transition_name | unnamed_trans) transition_condition
                'END_TRANSITION'
unnamed_trans ::= 'FROM' steps 'TO' steps
transition_name ::= identifier
steps ::= step_name | '(' step_name ',' step_name {',' step_name} ')'
transition_condition ::= ':' instruction_list | ':=' expression ';' | ':' (fbd_network | rung)
action ::= 'ACTION' action_name ':'
              function_block_body
            'END_ACTION'
```

SÉMANTIQUE: Voir 2.6. L'utilisation des réseaux de schémas en blocs fonctionnels et des échelons de schémas à contacts, respectivement désignés par les non terminaux *réseau_fbd* et *échelon*, pour l'expression des conditions de transition, doit être conforme aux prescriptions du paragraphe 2.6.

NOTE - La *liste_d'instructions* et l'*expression* non terminales sont définies respectivement en B.2.1 et B.3.1.

B.1.7 *Eléments de configuration*

RÈGLES DE PRODUCTION:

nom_de_configuration ::= identificateur

nom_du_type_de_ressource ::= identificateur

déclaration_de_configuration ::= 'CONFIGURATION' configuration_name
[global_var_declarations]
resource_declaration
{resource_declaration}
[access_declarations]
'END_CONFIGURATION'

déclaration_de_ressource ::= 'RESOURCE' resource_name 'ON' resource_type_name
[global_var_declarations]
{task_configuration ';'}
program_configuration ';'
{program_configuration ';'}
'END_RESOURCE'

nom_de_ressource ::= identificateur

déclarations_d'accès ::= 'VAR_ACCESS' access_declaration ';' {access_declaration ';'}
'END_VAR'

déclaration_d'accès ::= access_name ':' access_path ':' non_generic_type_name
[direction]

chemin_d'accès ::= [resource_name '.'] direct_variable
| resource_name '.' program_io_reference | global_var_reference

global_var_reference ::= [resource_name '.'] global_var_name
['.' structure_element_name]

nom_d'accès ::= identificateur

référence_io_programme ::= program_input_reference | program_output_reference

référence_de_sortie_du_programme ::= nom_du_programme '.' variable_symbolique

référence_d'entrée_du_programme ::= nom_du_programme '.' variable_symbolique

nom_du_programme ::= identificateur

autorisation ::= 'READ_WRITE'|'READ_ONLY'

configuration_de_tâche ::= 'TASK' task_name task_initialization

nom_de_tâche ::= identificateur

initialisation_de_tâche ::= '(' ['SINGLE' ':=' data source ','] ['INTERVAL' ':=' data_source ','] 'PRIORITY' ':=' integer')'

source_de_données ::= constant | global_var_reference | program_output_reference
| direct_variable

SEMANTICS: See 2.6. The use of function block diagram networks and ladder diagram rungs, denoted by the non-terminals *fbd_network* and *rung*, respectively, for the expression of transition conditions shall be as defined in 2.6.3.

NOTE - The non-terminals *instruction_list* and *expression* are defined in B.2.1 and B.3.1, respectively.

B.1.7 *Configuration elements*

PRODUCTION RULES:

```
configuration_name ::= identifier

resource_type_name ::= identifier

configuration_declaration ::= 'CONFIGURATION' configuration_name
                              [global_var_declarations]
                              resource_declaration
                              {resource_declaration}
                              [access_declarations]
                             'END_CONFIGURATION'

resource_declaration ::= 'RESOURCE' resource_name 'ON' resource_type_name
                         [global_var_declarations]
                         {task_configuration ';'}
                         program_configuration ';'
                         {program_configuration ';'}
                        'END_RESOURCE'

resource_name ::= identifier

access_declarations ::= 'VAR_ACCESS' access_declaration ';' {access_declaration ';'}
                        'END_VAR'

access_declaration ::= access_name ':' access_path ':' non_generic_type_name
    [direction]

access_path ::= [resource_name '.'] direct_variable
    | resource_name '.' program_io_reference | global_var_reference

global_var_reference ::= [resource_name '.'] global_var_name
                         ['.' structure_element_name]

access_name ::= identifier

program_io_reference ::= program_input_reference | program_output_reference

program_output_reference ::= program_name '.' symbolic_variable

program_input_reference ::= program_name '.' symbolic_variable

program_name ::= identifier

direction ::= 'READ_WRITE' | 'READ_ONLY'

task_configuration ::= 'TASK' task_name task_initialization

task_name ::= identifier

task_initialization ::= '(' ['SINGLE' ':=' data_source ','] ['INTERVAL' ':=' data_source ','
                        'PRIORITY' ':=' integer ')'

data_source ::= constant | global_var_reference | program_output_reference
                | direct_variable
```

configuration_du_programme ::= 'PROGRAM' program_name ['WITH' task_name] ':'
program_type_name ['(' prog_conf_elements ')']

prog_conf_elements ::= prog_conf_element {',' prog_conf_element}

prog_conf_element ::= fb_task | prog_cnxn

fb_task ::= fb_name 'WITH' task_name

prog_cnxn ::= symbolic_variable ':=' prog_data_source
| symbolic_variable '=>' data_sink

prog_data_source ::= constant | global_var_reference | direct_variable

data_sink ::= global_var_reference | direct_variable

SÉMANTIQUE: Voir 2.7.

B.2 Langage IL (liste d'instructions)

B.2.1 *Instructions et opérandes*

RÈGLES DE PRODUCTION:

instruction_list ::= instruction {instruction}

instruction ::= [[label ':'] (il_operation | il_fb_call)] EOL

label ::= identifier

il_operation ::= il_operator [' ' il_operand_list]

il_operand_list ::= il_operand (',' il_operand)

il_operand ::= [identifier ':='] (constant | variable)

il_fb_call ::= 'CAL' ['C'['N']] fb_name ['(' il_operand_list ')']

SÉMANTIQUE: Voir 3.2.

B.2.2 *Opérateurs*

RÈGLES DE PRODUCTION:

il_operator ::= ('LD' | 'ST') ['N'] |'S' | 'R'
| ('AND' | 'OR' | 'XOR') ['N'] ['(']
| ('ADD' | 'SUB' | 'MUL' | 'DIV') ['(']
| ('GT' | 'GE' | 'EQ' | 'LT' | 'LE') ['(']
| ('JMP' | 'RET') ['C' ['N']]
| 'S1' | 'R1' | 'CLK' | 'CU' | 'CD' | 'PV'
| 'IN' | 'PT' | ')'
| function_name

SÉMANTIQUE: Voir 3.2.

program_configuration ::= 'PROGRAM' program_name ['WITH' task_name] ':'
program_type_name ['(' prog_conf_elements ')']

prog_conf_elements ::= prog_conf_element {',' prog_conf_element}

prog_conf_element ::= fb_task | prog_cnxn

fb_task ::= fb_name 'WITH' task_name

prog_cnxn ::= symbolic_variable ':=' prog_data_source
| symbolic_variable '=>' data_sink

prog_data_source ::= constant | global_var_reference | direct_variable

data_sink ::= global_var_reference | direct_variable

SEMANTICS: See 2.7.

B.2 Language IL (Instruction List)

B.2.1 *Instructions and operands*

PRODUCTION RULES:

instruction_list ::= instruction {instruction}

instruction ::= [[label ':'] (il_operation | il_fb_call)] EOL

label ::= identifier

il_operation ::= il_operator [' ' il_operand_list]

il_operand_list ::= il_operand {',' il_operand}

il_operand ::= [identifier ':='] (constant | variable)

il_fb_call ::= 'CAL' ['C'['N']] fb_name ['(' il_operand_list ')']

SEMANTICS: See 3.2.

B.2.2 *Operators*

PRODUCTION RULES:

il_operator ::= ('LD' | 'ST') ['N'] | 'S' | 'R'
| ('AND' | 'OR' | 'XOR') ['N'] ['(']
| ('ADD' | 'SUB' | 'MUL' | 'DIV') ['(']
| ('GT' | 'GE' | 'EQ' | 'NE' | 'LT' | 'LE') ['(']
| ('JMP' | 'RET') ['C' ['N']]
|'S1' | 'R1' | 'CLK' | 'CU' | 'CD' | 'PV'
| ' IN' | 'PT' | ')'
| function_name

SEMANTICS: See 3.2.

B.3 Langage ST (Langage littéral structuré)

B.3.1 *Expressions*

RÈGLES DE PRODUCTION:

expression ::= xor_expression {'OR' xor_expression}

xor_expression ::= and_expression {'XOR' and_expression}

and_expression ::= comparison {('&' | 'AND') comparison}

comparison ::= add_expression {comparison_operator add_expression}

comparison_operator ::= '<' | '>' | '<=' | '>=' | '=' | '<>'

add_expression ::= term {add_operator term}

add_operator ::= '+' | '-'

term ::= power_expression {multiply_operator power_expression}

multiply_operator ::= '*' | '/' | 'MOD'

power_expression ::= unary_expression {'**' unary_expression}

unary_expression ::= [unary_operator] primary_expression

unary_operator ::= '-' | 'NOT'

primary_expression ::= constant | variable | '(' expression ')'
| function_name '(' [st_function_inputs] ')'

st_function_inputs ::= st_function_input { ',' st_function_input}

st_function_input ::= [variable_name ':='] expression

SÉMANTIQUE: Ces définitions ont été disposées pour mettre en évidence une déduction de haut en bas de la structure de l'expression. La précédence des opérations est de ce fait déduite d'une lecture de bas en haut des définitions des différents types d'expression. Pour des renseignements supplémentaires sur la sémantique de ces définitions, on se reportera à 3.3.1.

B.3.2 *Enoncés*

RÈGLE DE PRODUCTION:

statement_list ::= statement ';' {statement ';'}

statement ::= NIL | assignment_statement | subprogram_control_statement
| selection_statement | iteration_statement

SÉMANTIQUE: Voir 3.3.2.

B.3.2.1 *Enoncés d'affectation*

RÈGLE DE PRODUCTION:

assignment_statement ::= variable ':=' expression

SÉMANTIQUE: Voir 3.3.2.1.

B.3 Language ST (Structured Text)

B.3.1 *Expressions*

PRODUCTION RULES:

expression ::= xor_expression {'OR' xor_expression}

xor_expression ::= and_expression {'XOR' and_expression}

and_expression ::= comparison {('&' | 'AND') comparison}

comparison ::= add_expression {comparison_operator add_expression}

comparison_operator ::= '<' | '>' | '<=' | '>=' | '=' | '<>'

add_expression ::= term {add_operator term}

add_operator ::= '+' | '-'

term ::= power_expression {multiply_operator power_expression}

multiply_operator ::= '*' | '/' | 'MOD'

power_expression ::= unary_expression {'**' unary_expression}

unary_expression ::= [unary_operator] primary_expression

unary_operator ::= '-' | 'NOT'

primary_expression ::= constant | variable | '(' expression ')'
| function_name '(' [st_function_inputs] ')'

st_function_inputs ::= st_function_input {',' st_function_input}

st_function_input ::= [variable_name ':='] expression

SEMANTICS: These definitions have been arranged to show a top-down derivation of expression structure. The precedence of operations is then implied by a "bottom-up" reading of the definitions of the various kinds of expressions. Further discussion of the semantics of these definitions is given in 3.3.1.

B.3.2 *Statements*

PRODUCTION RULE:

statement_list ::= statement ';' {statement ';'}

statement ::= NIL | assignment_statement | subprogram_control_statement
| selection_statement | iteration_statement

SEMANTICS: See 3.3.2.

B.3.2.1 *Assignment statements*

PRODUCTION RULE:

assignment_statement ::= variable ':=' expression

SEMANTICS: See 3.3.2.1.

B.3.2.2 *Enoncés de commande de sous-programmes*

RÈGLE DE PRODUCTION:

subprogram_control_statement ::= fb_invocation | 'RETURN'

fb_invocation ::= fb_name '(' [fb_input_assignment {',' fb_input_assignment}] ')'

fb_input_assignment ::= variable_name ':=' expression

SÉMANTIQUE: Voir 3.3.2.2.

B.3.2.3 *Enoncés de sélection*

RÈGLE DE PRODUCTION:

selection_statement ::= if_statement | case_statement

if_statement ::= 'IF' expression 'THEN' statement_list
{'ELSIF' expression 'THEN' statement_list}
['ELSE' statement_list]
'END_IF'

case_statement ::= 'CASE' expression 'OF'
case_element {case_element}
['ELSE' statement_list]
'END_CASE'

case_element ::= case_list ':' statement_list

case_list ::= case_list_element {',' case_list_element}

case_list_element ::= subrange | signed_integer

SÉMANTIQUE : Voir 3.3.2.3.

B.3.2.4 *Enoncés d'itération*

RÈGLE DE PRODUCTION:

iteration_statement ::= for_statement | while_statement | repeat_statement | exit_statement

for_statement ::= 'FOR' control_variable ':=' for_list 'DO' statement_list 'END_FOR'

control_variable ::= identifier

for_list ::= expression 'TO' expression ['BY' expression]

while_statement ::= 'WHILE' expression 'DO' statement_list 'END_WHILE'

repeat_statement ::= 'REPEAT' statement_list 'UNTIL" expression 'END_REPEAT'

exit_statement ::= 'EXIT'

SÉMANTIQUE: Voir 3.3.2.4.

B.3.2.2 *Subprogram control statements*

PRODUCTION RULES:

subprogram_control_statement ::= fb_invocation | 'RETURN'

fb_invocation ::= fb_name '(' [fb_input_assignment {',' fb_input_assignment}] ')'

fb_input_assignment ::= variable_name ':=' expression

SEMANTICS: See 3.3.2.2.

B.3.2.3 *Selection statements*

PRODUCTION RULES:

selection_statement ::= if_statement | case_statement

if_statement ::= 'IF' expression 'THEN' statement_list
{'ELSIF' expression 'THEN' statement_list}
['ELSE' statement_list]
'END_IF'

case_statement ::= 'CASE' expression 'OF'
case_element {case_element}
['ELSE' statement_list]
'END_CASE'

case_element ::= case_list ':' statement_list

case_list ::= case_list_element {',' case_list_element}

case_list_element ::= subrange | signed_integer

SEMANTICS: See 3.3.2.3.

B.3.2.4 *Iteration statements*

PRODUCTION RULES:

iteration_statement ::= for_statement | while_statement | repeat_statement
| exit_statement

for_statement ::= 'FOR' control_variable ':=' for_list 'DO' statement_list 'END_FOR'

control_variable ::= identifier

for_list ::= expression 'TO' expression ['BY' expression]

while_statement ::= 'WHILE' expression 'DO' statement_list 'END_WHILE'

repeat_statement ::= 'REPEAT' statement_list 'UNTIL' expression 'END_REPEAT'

exit_statement ::= 'EXIT'

SEMANTICS: See 3.3.2.4.

Annexe C
(normative)

Délimiteurs et mots clés

L'utilisation des délimiteurs et mots clés dans la CEI 1131-3 est résumée aux tableaux C.1 et C.2. Les organisations de normalisation nationales peuvent publier des tables de traduction pour les parties textuelles des délimiteurs listés au tableau C.1 et des mots clés listés au tableau C.2.

Tableau C.1 – Délimiteurs

Délimiteurs	Paragraphe	Utilisation
Espace	2.1.4	Comme spécifié en 2.1.4.
(* *)	2.1.5	Début de commentaire Fin de commentaire
+	2.2.1 3.3.1	Signe en tête d'un libellé décimal Opérateur d'addition
–	2.2.1 2.2.3.2 3.3.1 4.1.1	Signe en tête d'un libellé décimal Séparateur année-mois-jour Opérateur de soustraction ou de négation Ligne horizontale
#	2.2.1 2.2.3	Séparateur de nombres basés Séparateur de libellés de temps
.	2.2.1 2.4.1.1 2.4.1.2 2.5.2.1	Séparateur entier/fraction Séparateur d'adresses hiérarchiques Séparateur d'éléments de structure Séparateur de structure de bloc fonctionnel
e ou E	2.2.1	Délimiteur d'exposant réel
'	2.2.2	Début et fin d'un cordon de caractères
$	2.2.2	Début d'un caractère spécial dans des cordons
2.2.3 – Délimiteurs de libellé de temps, comprenant: t#, T#, d, D, h, H, m, M, s, S, ms, MS DATE#, date#, D#, d#, TIME_OF_DAY#, time_of_day# TOD#, tod#, DATE_AND_TIME#, date_and_time#, DT#, dt#		
:	2.2.3.2 2.3.3.1 2.4.2 2.6.2 2.7 2.7 2.7 3.2.1 4.1.2	Séparateur de l'heure du jour Séparateur de spécification/nom du type Séparateur de variable/type Symbole terminal de nom de l'étape Séparateur de nom/type de RESOURCE Séparateur de nom/type de PROGRAM Séparateur de nom/chemin/type d'accès Symbole terminal d'étiquette d'instruction Symbole terminal de réseau
:=	2.3.3.1 2.7.1 3.3.2.1	Opérateur d'initialisation Opérateur de raccordement d'entrée Opérateur d'affectation

(suite à la page 300)

Annex C
(normative)

Delimiters and keywords

The usages of delimiters and keywords in IEC 1131-3 is summarized in tables C.1 and C.2. National standards organizations can publish tables of translations for the textual portions of the delimiters listed in table C.1 and the keywords listed in table C.2.

Table C.1 – Delimiters

Delimiters	Clause	Usage
Space	2.1.4	As specified in 2.1.4.
(* *)	2.1.5	Begin comment End comment
+	2.2.1 3.3.1	Leading sign of decimal literal Addition operator
–	2.2.1 2.2.3.2 3.3.1 4.1.1	Leading sign of decimal literal Year-month-day separator Subtraction, negation operator Horizontal line
#	2.2.1 2.2.3	Based number separator Time literal separator
.	2.2.1 2.4.1.1 2.4.1.2 2.5.2.1	Integer/fraction separator Hierarchical address separator Structure element separator Function block structure separator
e or E	2.2.1	Real exponent delimiter
'	2.2.2	Start and end of character string
$	2.2.2	Start of special character in strings
2.2.3 – Time literal delimiters, including: t#, T#, d, D, h, H, m, M, s, S, ms, MS DATE#, date#, D#, d#, TIME_OF_DAY#, time_of_day# TOD#, tod#, DATE_AND_TIME#, date_and_time#, DT#, dt#		
:	2.2.3.2 2.3.3.1 2.4.2 2.6.2 2.7 2.7 2.7 3.2.1 4.1.2	Time of day separator Type name/specification separator Variable/type separator Step name terminator RESOURCE name/type separator PROGRAM name/type separator Access name/path/type separator Instruction label terminator Network label terminator
:=	2.3.3.1 2.7.1 3.3.2.1	Initialization operator Input connection operator Assignment operator

(continued on page 301)

Tableau C.1 – Délimiteurs *(fin)*

Délimiteurs	Paragraphe	Utilisation
( )	2.3.3.1 2.3.3.1 2.4.1.2 2.4.2 2.4.2 3.2.2 3.3.1 3.3.1 3.3.2.2	Délimiteurs de liste d'énumération Délimiteurs d'intervalle Délimiteurs d'indice de tableau Délimiteurs de longueur de cordon Initialisation multiple Opérateur/modificateur de liste d'instructions Arguments de fonction Hiérarchie de sous-expression Délimiteurs de liste d'entrée de bloc fonctionnel
,	2.3.3.1 2.3.3.2 2.4.1 2.4.2 2.5.2.1 2.5.2.1 3.2.1 3.3.1 3.3.2.3	Séparateur de listes d'énumération Séparateur de valeurs initiales Séparateur d'indices de tableau Séparateur de variables déclarées Séparateur de valeurs initiales de bloc fonctionnel Séparateur de listes d'entrée de bloc fonctionnel Séparateur de listes d'opérandes Séparateur de listes d'arguments d'une fonction Séparateur de listes de valeurs de CASE
;	2.3.3.1 3.3	Séparateur de déclarations de type Séparateur d'énoncés
..	2.3.3.1 3.3.2.3	Séparateur d'intervalles Séparateur de plages de CASE
%	2.4.1.1	Préfixe de représentation directe
3.3.1 – Opérateurs infixés, comprenant: **, NOT, *, /, MOD, +, -, <, >, <=, >=, =, <>, &, AND, XOR, OR		
=>	2.7.1	Opérateur de raccordement de sortie
| ou !	4.1.1	Lignes verticales (note)
NOTE - "!" n'est autorisé que lorsque "|" n'existe pas dans une variante nationale du jeu de caractères.		

Table C.1 – *(concluded)*

Delimiters	Clause	Usage
( )	2.3.3.1 2.3.3.1 2.4.1.2 2.4.2 2.4.2 3.2.2 3.3.1 3.3.1 3.3.2.2	Enumeration list delimiters Subrange delimiters Array subscript delimiters String length delimiters Multiple initialization Instruction List modifier/operator Function arguments Subexpression hierarchy Function block input list delimiters
,	2.3.3.1 2.3.3.2 2.4.1 2.4.2 2.5.2.1 2.5.2.1 3.2.1 3.3.1 3.3.2.3	Enumeration list separator Initial value separator Array subscript separator Declared variable separator Function block initial value separator Function block input list separator Operand list separator Function argument list separator CASE value list separator
;	2.3.3.1 3.3	Type declaration separator Statement separator
..	2.3.3.1 3.3.2.3	Subrange separator CASE range separator
%	2.4.1.1	Direct representation prefix
3.3.1 – Infix operators, including: **, NOT, *, /, MOD, +, -, <, >, <=, >=, =, <>, &, AND, XOR, OR		
=>	2.7.1	Output connection operator
| or !	4.1.1	Vertical lines (note)
NOTE - "!" is only allowed when "|" does not exist in a national character set.		

Tableau C.2 – Mots clés

Mots clés	Paragraphe
Qualificatifs d'action	2.6.4.4
ACTION...END_ACTION	2.6.4.1
ARRAY...OF	2.3.3.1
AT	2.4.3
CASE...OF...ELSE...END_CASE	3.3.2.3
CONFIGURATION...END_CONFIGURATION	2.7.1
CONSTANT	2.4.3
Noms de types de données	2.3
EN, ENO	2.5.1.2
EXIT	3.3.2.4
FALSE	2.2.1
F_EDGE	2.5.2.2
FOR...TO...BY...DO...END_FOR	3.3.2.4
FUNCTION...END_FUNCTION	2.5.1.3
Noms de fonction	2.5.1
FUNCTION_BLOCK...END_FUNCTION_BLOCK	2.5.2.2
Noms de bloc fonctionnel	2.5.2
IF...THEN...ELSIF...ELSE...END_IF	3.3.2.3
INITIAL_STEP...END_STEP	2.6.2
PROGRAM...WITH...	2.7.1
PROGRAM...END_PROGRAM	2.5.3
R_EDGE	2.5.2.2
READ_ONLY, READ_WRITE	2.7.1
REPEAT...UNTIL...END_REPEAT	3.3.2.4
RESOURCE...ON...END_RESOURCE	2.7.1
RETAIN	2.4.3
RETURN	3.3.2.2
STEP...END_STEP	2.6.2
STRUCT...END_STRUCT	2.3.3.1
TASK	2.7.2
Opérateurs littéraux (langage IL)	3.2.2
(Langage ST)	3.3.1
TRANSITION...FROM...TO...END_TRANSITION	2.6.3
TRUE	2.2.1
TYPE...END_TYPE	2.3.3.1
VAR...END_VAR VAR_INPUT...END_VAR VAR_OUTPUT...END_VAR VAR_IN_OUT...END_VAR VAR_EXTERNAL...END_VAR	2.4.2
VAR_ACCESS...END_VAR	2.7.1
VAR_GLOBAL...END_VAR	2.7.1
WHILE...DO...END_WHILE	3.3.2.4
WITH	2.7.1

Table C.2 – Keywords

Keywords	Clause
Action qualifiers	2.6.4.4
ACTION...END_ACTION	2.6.4.1
ARRAY...OF	2.3.3.1
AT	2.4.3
CASE...OF...ELSE...END_CASE	3.3.2.3
CONFIGURATION...END_CONFIGURATION	2.7.1
CONSTANT	2.4.3
Data type names	2.3
EN, ENO	2.5.1.2
EXIT	3.3.2.4
FALSE	2.2.1
F_EDGE	2.5.2.2
FOR...TO...BY...DO...END_FOR	3.3.2.4
FUNCTION...END_FUNCTION	2.5.1.3
Function names	2.5.1
FUNCTION_BLOCK...END_FUNCTION_BLOCK	2.5.2.2
Function Block names	2.5.2
IF...THEN...ELSIF...ELSE...END_IF	3.3.2.3
INITIAL_STEP...END_STEP	2.6.2
PROGRAM...WITH...	2.7.1
PROGRAM...END_PROGRAM	2.5.3
R_EDGE	2.5.2.2
READ_ONLY, READ_WRITE	2.7.1
REPEAT...UNTIL...END_REPEAT	3.3.2.4
RESOURCE...ON...END_RESOURCE	2.7.1
RETAIN	2.4.3
RETURN	3.3.2.2
STEP...END_STEP	2.6.2
STRUCT...END_STRUCT	2.3.3.1
TASK	2.7.2
Textual operators (IL language)	3.2.2
(ST language)	3.3.1
TRANSITION...FROM...TO...END_TRANSITION	2.6.3
TRUE	2.2.1
TYPE...END_TYPE	2.3.3.1
VAR...END_VAR VAR_INPUT...END_VAR VAR_OUTPUT...END_VAR VAR_IN_OUT...END_VAR VAR_EXTERNAL...END_VAR	2.4.2
VAR_ACCESS...END_VAR	2.7.1
VAR_GLOBAL...END_VAR	2.7.1
WHILE...DO...END_WHILE	3.3.2.4
WITH	2.7.1

Annexe D
(normative)

Paramètres dépendant de l'implémentation

Les paramètres dépendant de l'implémentation définis dans la CEI 1131-3, et le paragraphe de référence primaire pour chacun sont donnés au tableau D.1.

Tableau D.1 – Paramètres dépendant de l'implémentation

Paragraphe	Paramètre
1.5.1	Procédures de gestion d'une erreur
2.1.1	Caractères nationaux utilisés Signe # ou "livre Sterling" Signe $ ou "devise" | ou !
2.1.2	Longueur maximale des identificateurs
2.1.5	Longueur maximale du commentaire
2.2.3.1	Plage des valeurs de durée
2.3.1	Plage des valeurs pour les variables de type TIME Précision de représentation des secondes dans les types TIME_OF_DAY et DATE_AND_TIME
2.3.3	Nombre maximal d'indices de tableau Taille maximale des tableaux Nombre maximal d'éléments de structure Taille maximale des structures Nombre maximal de variables par déclaration
2.3.3.1	Nombre maximal de valeurs énumérées
2.3.3.2	Longueur maximale par défaut des variables STRING (cordon) Longueur maximale autorisée pour les variables STRING
2.4.1.1	Nombre maximal de niveaux hiérarchiques Configuration logique ou physique
2.4.1.2	Nombre maximal d'indices Nombre maximal de valeurs d'indice Nombre maximal de niveaux de structures
2.4.2	Initialisation d'entrées de système
2.4.3	Nombre maximal de variables par déclaration
2.5	Informations nécessaires à la détermination des temps d'exécution des unités d'organisation de programme
2.5.1.1	Méthode de représentation des fonctions (noms ou symboles)
2.5.1.3	Nombre maximal de spécifications fonctionnelles
2.5.1.5	Nombre maximal d'entrées de fonctions extensibles
2.5.1.5.1	Effets des conversions de type sur la précision
2.5.1.5.2	Précision des fonctions d'une variable Implémentation des fonctions arithmétiques

(suite à la page 306)

Annex D
(normative)

Implementation-dependent parameters

The implementation-dependent parameters defined in IEC 1131-3, and the primary reference clause for each, are listed in table D.1.

Table D.1 – Implementation-dependent parameters

Clause	Parameters
1.5.1	Error handling procedures
2.1.1	National characters used # or "pounds Sterling" sign $ or "currency" sign | or !
2.1.2	Maximum length of identifiers
2.1.5	Maximum comment length
2.2.3.1	Range of values of duration
2.3.1	Range of values for variables of type TIME Precision of representation of seconds in types TIME_OF_DAY and DATE_AND_TIME
2.3.3	Maximum number of array subscripts Maximum array size Maximum number of structure elements Maximum structure size Maximum number of variables per declaration
2.3.3.1	Maximum number of enumerated values
2.3.3.2	Default maximum length of STRING variables Maximum allowed length of STRING variables
2.4.1.1	Maximum number of hierarchical levels Logical or physical mapping
2.4.1.2	Maximum number of subscripts Maximum range of subscript values Maximum number of levels of structures
2.4.2	Initialization of system inputs
2.4.3	Maximum number of variables per declaration
2.5	Information to determine execution times of program organization units
2.5.1.1	Method of function representation (names or symbols)
2.5.1.3	Maximum number of function specifications
2.5.1.5	Maximum number of inputs of extensible functions
2.5.1.5.1	Effects of type conversions on accuracy
2.5.1.5.2	Accuracy of functions of one variable Implementation of arithmetic functions

(continued on page 307)

Tableau D.1 *(fin)*

Paragraphe	Paramètre
2.5.2	Nombre maximal de spécifications et d'instanciations de blocs fonctionnels
2.5.2.3.3	PV minimal, PV maximal des compteurs
2.5.3	Limites de la taille des programmes
2.6	Effets du séquencement et de la portabilité des éléments de commande d'exécution
2.6.2	Précision de mesure du temps écoulé sur une étape Nombre maximal d'étapes par diagramme fonctionnel en séquence
2.6.3	Nombre maximal de transitions par diagramme fonctionnel en séquence et par étape
2.6.4	Mécanisme de commande d'action
2.6.4.2	Nombre maximal de blocs d'action par étape
2.6.5	Indication graphique de l'état des étapes Temps d'effacement des transitions Largeur maximale des constructions de divergence/de convergence
2.7.1	Contenus des bibliothèques RESOURCE
2.7.2	Nombre maximal de tâches Résolution des intervalles de tâche Ordonnancement préemptif ou non préemptif
3.3.1	Longueur maximale des expressions Evaluation partielle des expressions booléennes
3.3.2	Longueur maximale des énoncés
3.3.2.3	Nombre maximal de sélections de CASE
3.3.2.4	Valeur de la variable de commande à la fin de la boucle FOR
4.1.1	Représentation graphique/semi-graphique Restrictions sur la topologie des réseaux
4.1.3	Ordre d'évaluation des boucles d'asservissement

Table D.1 *(concluded)*

Clause	Parameters
2.5.2	Maximum number of function block specifications and instantiations
2.5.2.3.3	PVmin, PVmax of counters
2.5.3	Program size limitations
2.6	Timing and portability effects of execution control elements
2.6.2	Precision of step elapsed time Maximum number of steps per SFC
2.6.3	Maximum number of transitions per SFC and per step
2.6.4	Action control mechanism
2.6.4.2	Maximum number of action blocks per step
2.6.5	Graphic indication of step state Transition clearing time Maximum width of diverge/converge constructs
2.7.1	Contents of RESOURCE libraries
2.7.2	Maximum number of tasks Task interval resolution Pre-emptive or non-pre-emptive scheduling
3.3.1	Maximum length of expressions Partial evaluation of Boolean expressions
3.3.2	Maximum length of statements
3.3.2.3	Maximum number of CASE selections
3.3.2.4	Value of control variable upon termination of FOR loop
4.1.1	Graphic/semigraphic representation Restrictions on network topology
4.1.3	Evaluation order of feedback loops

Annexe E
(normative)

Situations d'erreur

Les situations d'erreur définies dans la CEI 1131-3 ainsi que les paragraphes dans lesquels elles sont décrites sont énumérés dans le tableau E.1. Ces erreurs peuvent être détectées au cours de la préparation du programme à exécuter ou pendant son exécution. Le fabricant doit préciser les dispositions à prendre lorsque ces erreurs sont détectées, conformément aux prescriptions de 1.5.1.

Tableau E.1 – Situations d'erreurs

Paragraphe	Situations d'erreurs
2.3.3.1	La valeur d'une variable dépasse l'intervalle spécifié
2.4.2	La longueur de la liste d'initialisation ne correspond pas au nombre d'entrées du tableau
2.5.1.5.1	Erreurs de conversion de type
2.5.1.5.2	Le résultat numérique dépasse la plage pour le type de donnée Division par zéro
2.5.1.5.4	Mélange des types de données en entrée d'une fonction de sélection Sélecteur (K) en dehors de la plage pour la fonction MUX
2.5.1.5.5	Position de caractère spécifiée invalide Le résultat dépasse la longueur maximale autorisée pour le cordon
2.5.1.5.6	Le résultat dépasse la plage pour ce type de donnée
2.6.2	Zéro ou plus d'une étape initiale dans un réseau diagramme fonctionnel en séquence Le programme utilisateur tente de modifier l'état ou le temps d'étape
2.6.2.5	Transitions simultanément vraies et non prioritaires dans une divergence de sélection
2.6.3	Effets indésirables dans l'évaluation d'une situation de transition
2.6.4.5	Erreur d'encombrement de commande d'action
2.6.5	Schéma diagramme fonctionnel en séquence "incertain" ou "inaccessible"
2.7.1	Conflit de type de donnée dans VAR_ACCESS
2.7.2	Les tâches exigent trop de ressources de traitement Délais d'exécution non respectés Autres conflits d'ordonnancement de tâches
3.2.2	Le résultat numérique dépasse la plage pour ce type de donnée
3.3.1	Division par zéro Type de donnée invalide pour l'opération
3.3.2.1	Retour d'une fonction sans valeur assignée
3.3.2.4	Boucle continue
4.1.1	Même identificateur utilisé comme étiquette de connecteur et nom d'élément
4.1.4	Variable d'asservissement non initialisée
4.1.5	Comme pour 2.5.1.5.2

Annex E
(normative)

Error conditions

The error conditions defined in IEC 1131-3, and the primary reference clause for each, are listed in table E.1. These errors may be detected during preparation of the program for execution or during execution of the program. The manufacturer shall specify the disposition of these errors according to the provisions of subclause 1.5.1 of this part.

Table E.1 – Error conditions

Clause	Errors conditions
2.3.3.1	Value of a variable exceeds the specified subrange
2.4.2	Length of initialization list does not match number of array entries
2.5.1.5.1	Type conversion errors
2.5.1.5.2	Numerical result exceeds range for data type Division by zero
2.5.1.5.4	Mixed input data types to a selection function Selector (K) out of range for MUX function
2.5.1.5.5	Invalid character position specified Result exceeds maximum string length
2.5.1.5.6	Result exceeds range for data type
2.6.2	Zero or more than one initial steps in SFC network User program attempts to modify step state or time
2.6.2.5	Simultaneously true, non-prioritized transitions in a selection divergence
2.6.3	Side effects in evaluation of transition condition
2.6.4.5	Action control contention error
2.6.5	"Unsafe" or "unreachable" SFC
2.7.1	Data type conflict in VAR_ACCESS
2.7.2	Tasks require too many processor resources Execution deadline not met Other task scheduling conflicts
3.2.2	Numerical result exceeds range for data type
3.3.1	Division by zero Invalid data type for operation
3.3.2.1	Return from function without value assigned
3.3.2.4	Iteration fails to terminate
4.1.1	Same identifier used as connector label and element name
4.1.4	Uninitialized feedback variable
4.1.5	As for 2.5.1.5.2

Annexe F
(informative)

Exemples

F.1 Fonction WEIGH

La fonction WEIGH utilisée dans cet exemple met en oeuvre les fonctions de conversion BCD/binaire d'une entrée poids brut obtenu grâce à une bascule, la soustraction en entier binaire d'une tare préalablement convertie et stockée dans la mémoire de l'automate programmable, et la conversion du poids obtenu comme résultat en format BCD, en vue, par exemple, de son affichage. L'entrée "EN" est utilisée pour signaler que la bascule est prête pour l'opération de pesée.

La sortie "ENO" indique qu'une commande appropriée existe (par exemple, par l'intermédiaire d'un bouton poussoir), que la bascule est en état d'effectuer la pesée et de l'afficher, et que chaque fonction se déroule correctement.

La déclaration sous forme littérale de cette fonction se présente de la manière suivante:

```
FUNCTION WEIGH : WORD (* BCD encoded *)
  VAR_INPUT (* "EN" input is used to indicate "scale ready" *)
    weigh_command : BOOL ;
    gross_weight : WORD ; (* BCD encoded *)
    tare_weight : INT ;
  END_VAR

(* Function Body *)

END_FUNCTION                (* Implicit "ENO" *)
```

Le corps de la fonction WEIGH en langage IL est le suivant:

```
          LD          weigh_command
          JMPC        WEIGH_NOW
          ST          ENO             (* No weighing, 0 to "ENO" *)
          RET
WEIGH_NOW: LD         gross_weight
          BCD_TO_INT
          SUB         tare_weight
          INT_TO_BCD                  (* Return evaluated weight *)
```

Le corps de la fonction WEIGH en langage ST est le suivant:

```
IF weight_command THEN
  WEIGH  := INT_TO_BCD (BCD_TO INT (gross_weight) - tare_weight) ;
END_IF ;
```

Annex F
(informative)

Examples

F.1 Function WEIGH

Example function WEIGH provides the functions of BCD-to-binary conversion of a gross-weight input from a scale, the binary integer subtraction of a tare weight which has been previously converted and stored in the memory of the programmable controller, and the conversion of the resulting net weight back to BCD form, e.g., for an output display. The "EN" input is used to indicate that the scale is ready to perform the weighing operation.

The "ENO" output indicates that an appropriate command exists (e.g., from an operator pushbutton), the scale is in proper condition for the weight to be read, and each function has a correct result.

A textual form of the declaration of this function is:

```
FUNCTION WEIGH : WORD (* BCD encoded *)
  VAR_INPUT (* "EN" input is used to indicate "scale ready" *)
    weigh_command : BOOL ;
    gross_weight : WORD ; (* BCD encoded *)
    tare_weight : INT ;
  END_VAR
(* Function Body *)
END_FUNCTION                    (* Implicit "ENO" *)
```

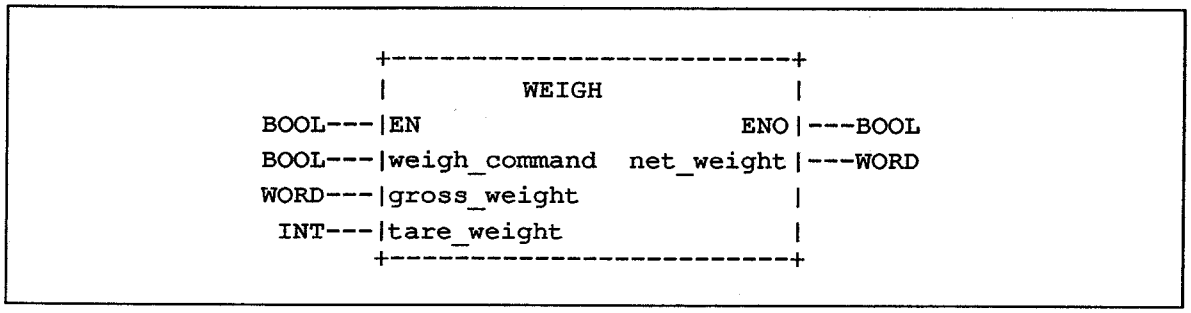
The body of function WEIGH in the IL language is:

```
          LD          weigh_command
          JMPC        WEIGH_NOW
          ST          ENO              (* No weighing, 0 to "ENO" *)
          RET
WEIGH_NOW: LD         gross_weight
          BCD_TO_INT
          SUB         tare_weight
          INT_TO_BCD                   (* Return evaluated weight *)
```

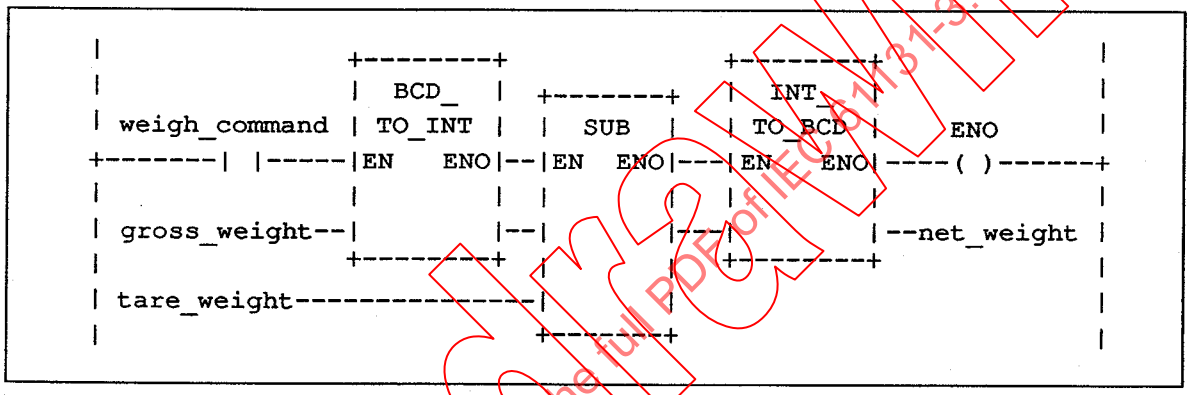
The body of function WEIGH in the ST language is:

```
IF weight_command THEN
  WEIGH  := INT_TO_BCD (BCD_TO INT (gross_weight) - tare_weight) ;
END_IF ;
```

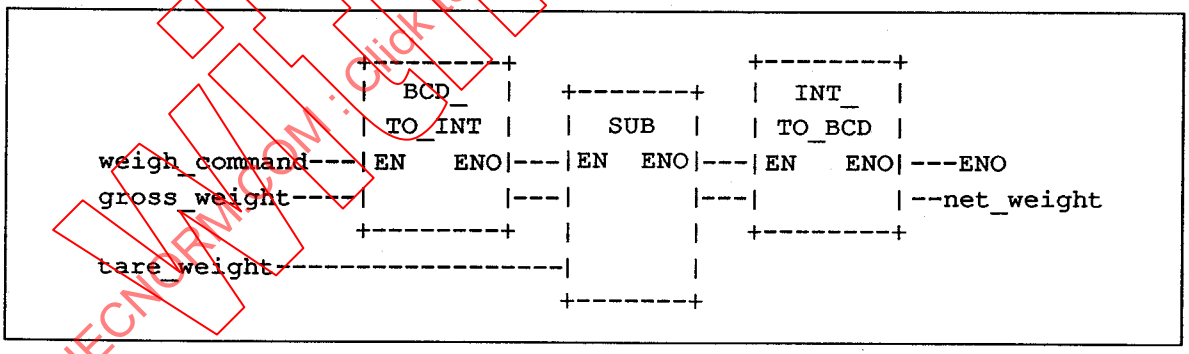
Une déclaration graphique équivalente de la fonction WEIGH est la suivante:



Le corps de la fonction en langage LD est le suivant:



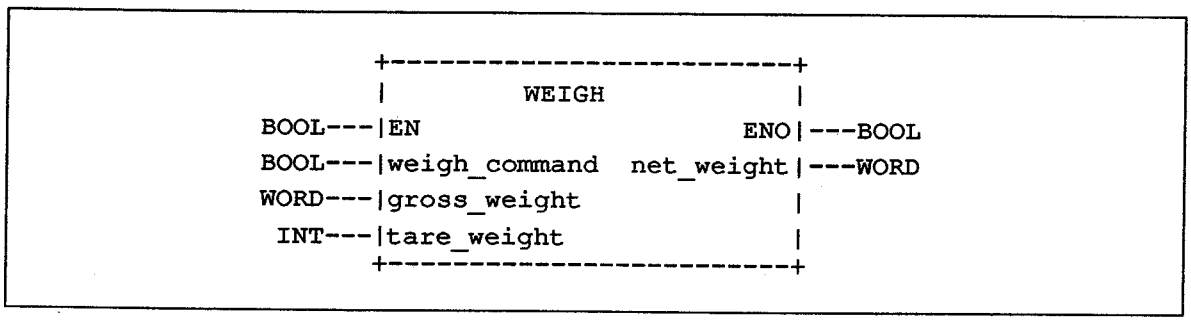
Le corps de la fonction en langage FBD est le suivant:



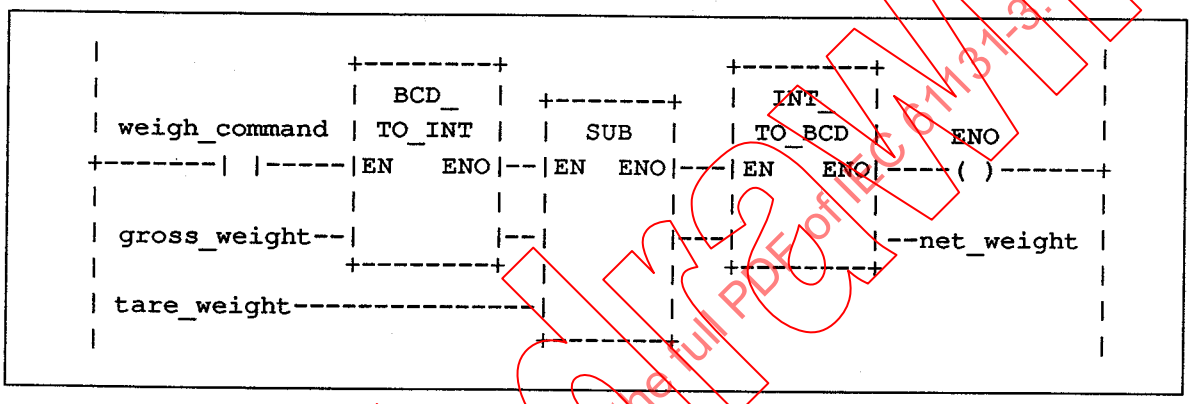
F.2 Bloc Fonctionnel CMD_MONITOR

L'exemple utilisant le bloc fonctionnel CMD_MONITOR illustre la commande d'un ensemble d'exploitation capable de répondre à une commande booléenne (la sortie CMD) et de renvoyer un signal d'asservissement booléen (l'entrée FDBK) qui indique la bonne exécution de l'action demandée. Le bloc fonctionnel permet la commande manuelle, par l'intermédiaire de l'entrée MAN_CMD, ou la commande automatisée, par l'intermédiaire de l'entrée AUTO_CMD, suivant l'état de l'entrée AUTO_MODE (respectivement 0 ou 1). La vérification de l'entrée MAN_CMD s'effectue par l'intermédiaire de l'entrée MAN_CMD_CHK, qui doit être à 0 pour que l'entrée MAN_CMD soit validée.

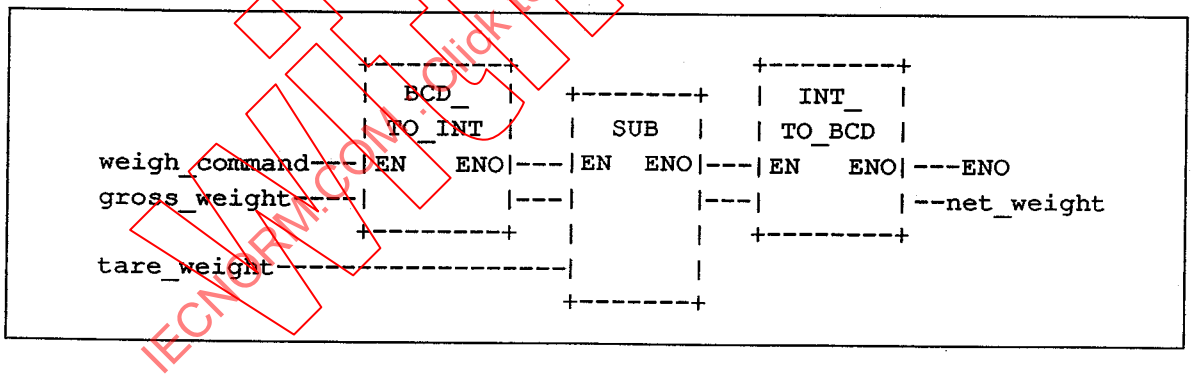
An equivalent graphical declaration of function WEIGH is:



The function body in the LD language is:



The function body in the FBD language is:



F.2 Function block CMD_MONITOR

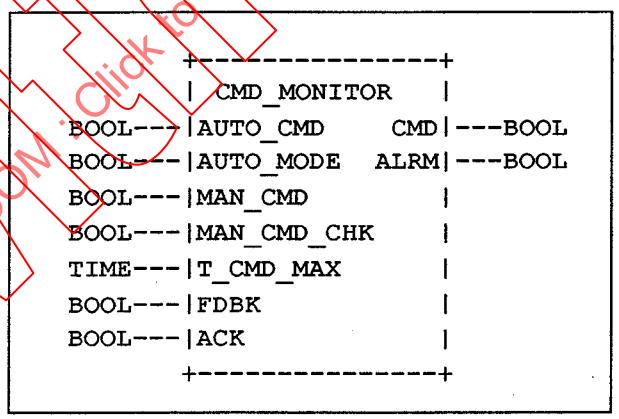
Example function block CMD_MONITOR illustrates the control of an operative unit which is capable of responding to a Boolean command (the CMD output) and returning a Boolean feedback signal (the FDBK input) indicating successful completion of the commanded action. The function block provides for manual control via the MAN_CMD input, or automated control via the AUTO_CMD input, depending on the state of the AUTO_MODE input (0 or 1 respectively). Verification of the MAN_CMD input is provided via the MAN_CMD_CHK input, which must be 0 in order to enable the MAN_CMD input.

Si aucune confirmation de la bonne exécution de la commande n'est reçue sur l'entrée FDBK dans un délai déterminé par l'entrée T_CMD_MAX, la commande est annulée et un état d'alarme est signalé par l'intermédiaire de la sortie ALRM. L'état d'alarme peut être annulé par l'entrée ACK (acquittement), ce qui permet la poursuite du cycle de commande.

La déclaration sous forme littérale de ce bloc fonctionnel est la suivante:

```
FUNCTION_BLOCK CMD_MONITOR
  VAR_INPUT AUTO_CMD : BOOL ;   (* Automated command *)
  AUTO_MODE : BOOL ;            (* AUTO_CMD enable *)
  MAN_CMD : BOOL ;              (* Manual command *)
  MAN_CMD_CHK : BOOL ;          (* Negated MAN_CMD to debounce *)
  T_CMD_MAX : TIME ;            (* Max time from CMD to FDBK *)
  FDBK : BOOL ;                 (* Confirmation of CMD completion by operative unit *)
  ACK : BOOL ;                  (* Acknowledge/cancel ALRM *)
END_VAR
VAR_OUTPUT CMD : BOOL ;         (* Command to operative unit *)
  ALRM : BOOL ;                 (* T_CMD_MAX expired without FDBK *)
END_VAR
VAR CMD_TMR : TON ;             (* CMD-to-FDBK timer *)
  ALRM_FF : SR ;                (* Note over-riding "S" input: *)
END_VAR                         (* Command must be cancelled before "ACK" can cancel alarm *)
(* Function Block Body *)
END_FUNCTION_BLOCK
```

Une déclaration graphique équivalente est:



Le corps du bloc fonctionnel CMD_MONITOR dans le langage ST est le suivant:

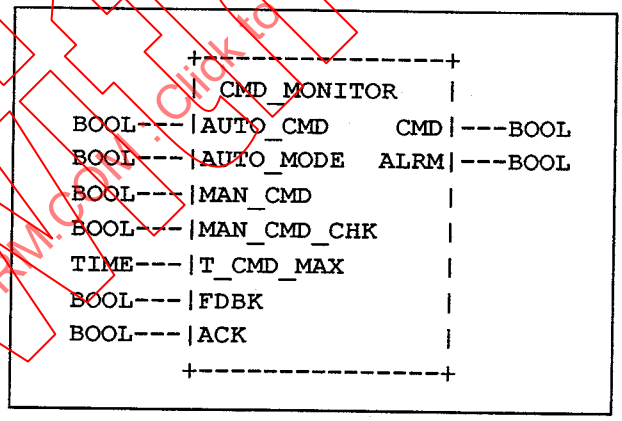
```
CMD := AUTO_CMD & AUTO_MODE
       OR MAN_CMD & NOT MAN_CMD_CHK & NOT AUTO_MODE ;
CMD_TMR (IN := CMD, PT := T_CMD_MAX) ;
ALRM_FF (S1 := CMD_TMR.Q & NOT FDBK, R := ACK) ;
ALRM := ALRM_FF.Q1 ;
```

If confirmation of command completion is not received on the FDBK input within a predetermined time specified by the T_CMD_MAX input, the command is cancelled and an alarm condition is signalled via the ALRM output. The alarm condition may be cancelled by the ACK (acknowledge) input, enabling further operation of the command cycle.

A textual form of the declaration of function block CMD_MONITOR is:

```
FUNCTION_BLOCK CMD_MONITOR
   VAR_INPUT AUTO_CMD : BOOL ;   (* Automated command *)
   AUTO_MODE : BOOL ;            (* AUTO_CMD enable *)
   MAN_CMD : BOOL ;              (* Manual command *)
   MAN_CMD_CHK : BOOL ;          (* Negated MAN_CMD to debounce *)
   T_CMD_MAX : TIME ;            (* Max time from CMD to FDBK *)
   FDBK : BOOL ;                 (* Confirmation of CMD completion by operative unit *)
   ACK : BOOL ;                  (* Acknowledge/cancel ALRM *)
END_VAR

VAR_OUTPUT CMD : BOOL ;          (* Command to operative unit *)
   ALRM : BOOL ;                 (* T_CMD_MAX expired without FDBK *)
END_VAR

VAR CMD_TMR : TON ;              (* CMD-to-FDBK timer *)
   ALRM_FF : SR ;                (* Note over-riding "S" input: *)
END_VAR                          (* Command must be cancelled before "ACK" can cancel alarm *)

(* Function Block Body *)

END_FUNCTION_BLOCK
```

An equivalent graphical declaration is:



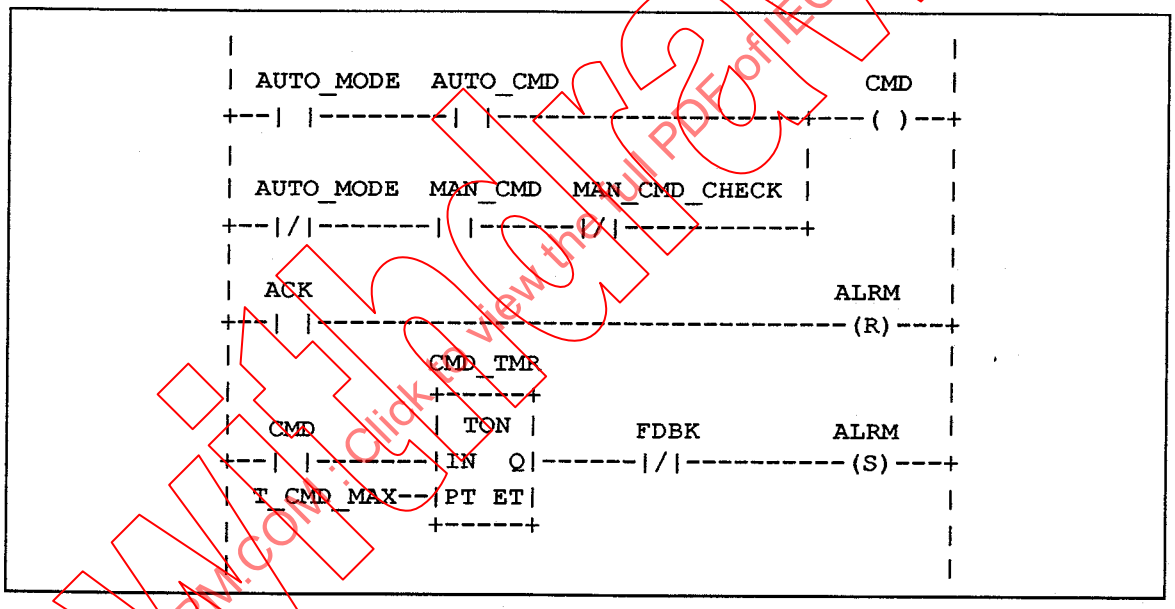
The body of function block CMD_MONITOR in the ST language is:

```
CMD := AUTO_CMD & AUTO_MODE
       OR MAN_CMD & NOT MAN_CMD_CHK & NOT AUTO_MODE ;
CMD_TMR (IN := CMD, PT := T_CMD_MAX) ;
ALRM_FF (S1 := CMD_TMR.Q & NOT FDBK, R := ACK) ;
ALRM := ALRM_FF.Q1 ;
```

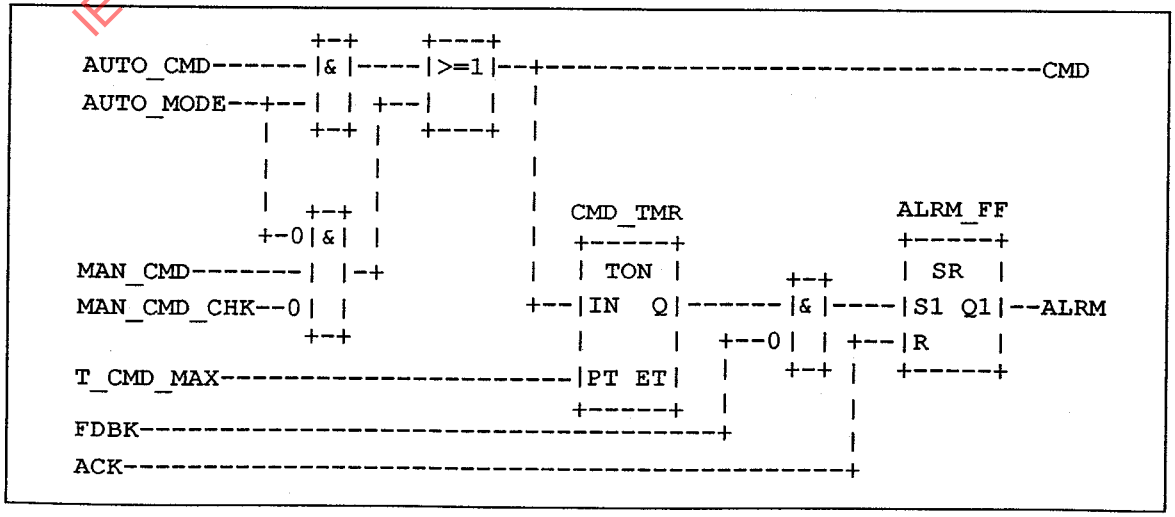
Le corps du bloc fonctionnel CMD_MONITOR dans le langage IL est le suivant:

```
LD      T_CMD_MAX
ST      CMD_TMR.PT          (* Store an input to the TON FB *)
LD      AUTO_CMD
AND     AUTO_MODE
OR(     MAN_CMD
ANDN    AUTO_MODE
ANDN    MAN_CMD_CHK
)
ST      CMD
IN      CMD_TMR             (* Invoke the TON FB *)
LD      CMD_TMR.Q
ANDN    FDBK
ST      ALRM_FF.S1          (* Store an input to the SR FB *)
LD      ACK
R       ALRM_FF             (* Invoke the SR FB *)
LD      ALRM_FF.Q1
ST      ALRM
```

Le corps du bloc fonctionnel CMD_MONITOR dans le langage LD est le suivant:



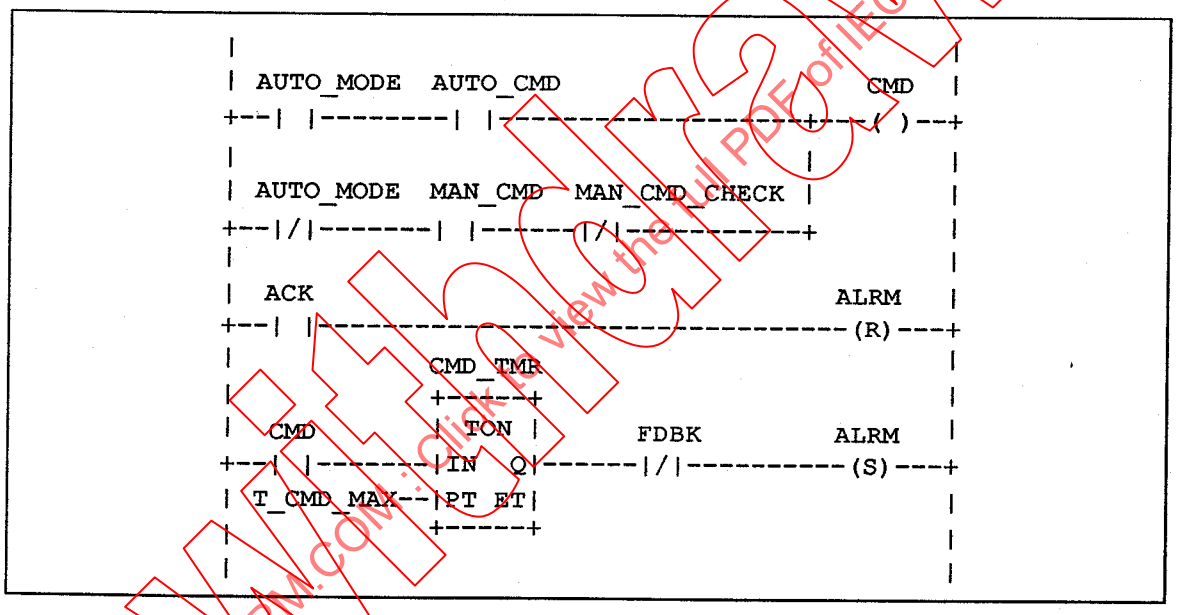
Le corps du bloc fonctionnel CMD_MONITOR dans le langage FBD est le suivant:



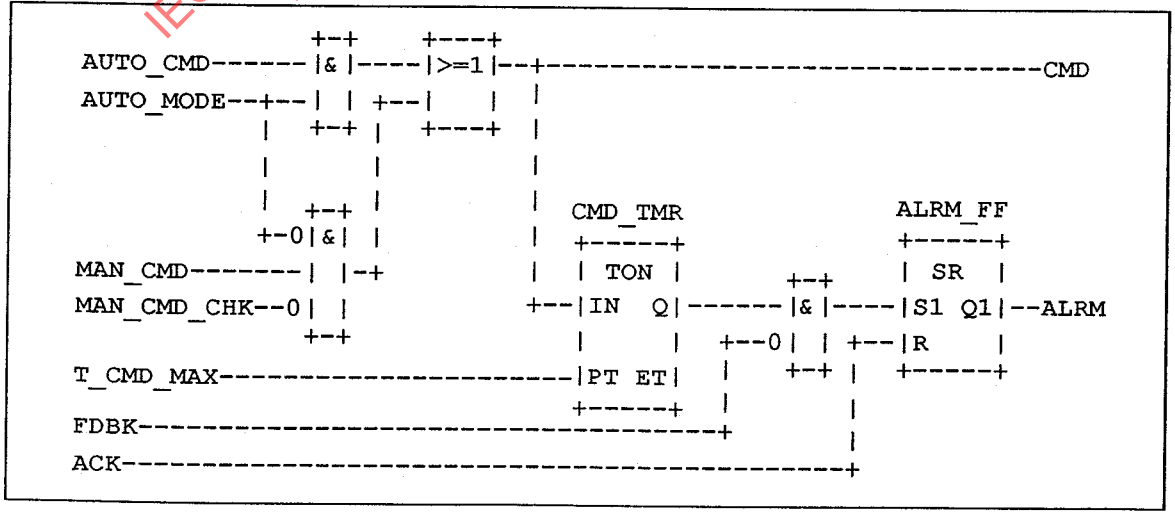
The body of function block CMD_MONITOR in the IL language is:

```
LD      T_CMD_MAX
ST      CMD_TMR.PT          (* Store an input to the TON FB *)
LD      AUTO_CMD
AND     AUTO_MODE
OR(     MAN_CMD
ANDN    AUTO_MODE
ANDN    MAN_CMD_CHK
)
ST      CMD
IN      CMD_TMR             (* Invoke the TON FB *)
LD      CMD_TMR.Q
ANDN    FDBK
ST      ALRM_FF.S1          (* Store an input to the SR FB *)
LD      ACK
R       ALRM_FF             (* Invoke the SR FB *)
LD      ALRM_FF.Q1
ST      ALRM
```

The body of function block CMD_MONITOR in the LD language is:



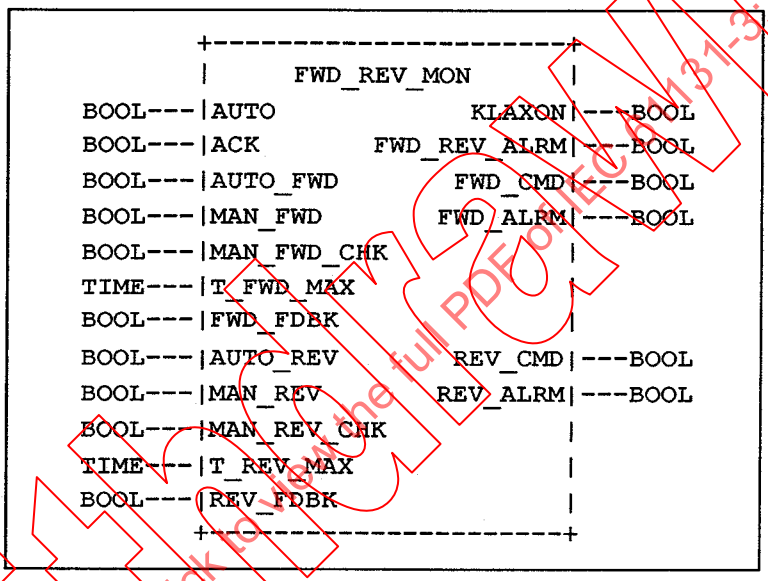
The body of function block CMD_MONITOR in the FBD language is:



F.3 Bloc fonctionnel FWD_REV_MON

Le bloc fonctionnel FWD_REV_MON utilisé comme exemple illustre la commande d'un équipement capable d'une action de positionnement bidirectionnel, par exemple, une vanne motorisée. Deux modes de commande (automatique ou manuelle) sont possibles, et des dispositifs d'alarme sont prévus pour chaque sens de déplacement, comme décrit ci-dessus pour le bloc fonctionnel CMD_MONITOR. En outre, toute collision entre les commandes de déplacement vers l'avant et vers l'arrière entraîne l'annulation des deux commandes et la signalisation d'un état d'alarme. Le OR booléen de tous les états d'alarme est disponible sur la sortie KLAXON afin d'avertir l'opérateur.

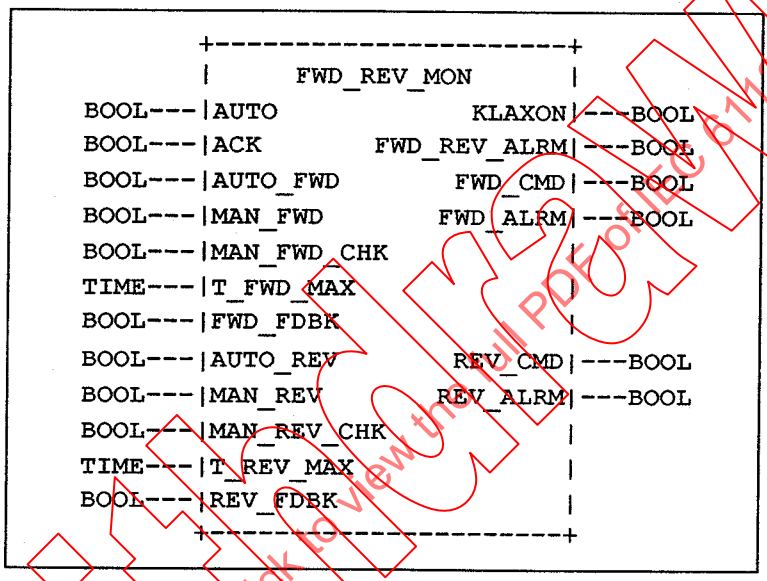
Une déclaration graphique de ce bloc fonctionnel est la suivante:



F.3 Function block FWD_REV_MON

Example function block FWD_REV_MON illustrates the control of an operative unit capable of two-way positioning action, e.g., a motor-operated valve. Both automated and manual control modes are possible, with alarm capabilities provided for each direction of motion, as described for function block CMD_MONITOR above. In addition, contention between forward and reverse commands causes the cancellation of both commands and signalling of an alarm condition. The Boolean OR of all alarm conditions is made available as a KLAXON output for operator signaling.

A graphical declaration of this function block is:



La déclaration littérale de la déclaration du bloc fonctionnel FWD_REV_MON est la suivante:

```
FUNCTION_BLOCK FWD_REV_MON
   VAR_INPUT AUTO : BOOL ;              (* Enable automated commands *)
   ACK : BOOL ;                         (* Acknowledge/cancel all alarms *)
   AUTO_FWD : BOOL ;                    (* Automated forward command *)
   MAN_FWD : BOOL ;                     (* Manual forward command *)
   MAN_FXWD_CHK : BOOL ;                (* Negated MAN_FWD for debouncing *)
   T_FWD_MAX : TIME ;                   (* Maximum time from FWD_CMD to FWD_FDBK *)
   FWD_FDBK : BOOL ;                    (* Confirmation of FWD_CMD completion *)
                                        (* by operative unit *)
   AUTO_REV : BOOL ;                    (* Automated reverse command *)
   MAN_REV : BOOL ;                     (* Manual reverse command *)
   MAN_REV_CHK : BOOL ;                 (* Negated MAN_REV for debouncing *)
   T_REV_MAX : TIME ;                   (* Maximum time from REV_CMD to REV_FDBK *)
   REV_FDBK : BOOL ;                    (* Confirmation of REV_CMD completion *)
END_VAR                                 (* by operative unit *)

VAR_OUTPUT KLAXON : BOOL ;              (* Any alarm active *)
   FWD_REV_ALRM : BOOL ;                (* Forward/reverse command conflict *)
   FWD_CMD : BOOL ;                     (* "Forward" command to operative unit *)
   FWD_ALRM : BOOL ;                    (* T_FWD_MAX expired without FWD_FDBK *)
   REV_CMD : BOOL ;                     (* "Reverse" command to operative unit *)
   REV_ALRM : BOOL ;                    (* T_REV_MAX expired without REV_FDBK *)
END_VAR

VAR FWD_MON : CMD_MONITOR ;             (* "Forward" command monitor *)
   REV_MON : CMD_MONITOR ;              (* "Reverse" command monitor *)
   FWD_REV_FF : SR ;                    (* Forward/reverse contention latch *)
END_VAR

(* Function block body *)
END_FUNCTION_BLOCK
```

Le corps du bloc fonctionnel FWD_REV_MON peut être écrit comme suit dans le langage ST:

```
(* Evaluate internal function blocks *)

FWD_MON          (AUTO_MODE := AUTO,
                  ACK := ACK,
                  AUTO_CMD := AUTO_FWD,
                  MAN_CMD := MAN_FWD,
                  MAN_CMD_CHK := MAN_FWD_CHK,
                  T_CMD_MAX := T_FWD_MAX,
                  FDBK := FWD_FDBK) ;
REV_MON          (AUTO MODE := AUTO,
                  ACK := ACK,
                  AUTO_CMD := AUTO_REV,
                  MAN_CMD := MAN_REV,
                  MAN_CMD_CHK := MAN_REV_CHK,
                  T_CMD_MAX := T_REV_MAX,
                  FDBK := REV_FDBK) ;
FWD_REV_FF       (S1 := FWD_MON.CMD & REV_MON.CMD, R := ACK) ;

(* Transfer data to outputs *)

FWD_REV_ALRM := FWD_REV_FF.Q1 ;
FWD_CMD := FWD_MON.CMD & NOT FWD_REV_ALRM ;
FWD_ ALRM := FWD_MON.ALRM ;
REV_CMD := REV_MON.CMD & NOT FWD_REV_ALRM ;
REV_ALRM := REV_MON.ALRM ;
KLAXON := FWD_ALRM OR REV_ALRM OR FWD_REV_ALRM ;
```

A textual form of the declaration of function block FWD_REV_MON is:

```
FUNCTION_BLOCK FWD_REV_MON
   VAR_INPUT AUTO : BOOL ;              (* Enable automated commands *)
   ACK : BOOL ;                         (* Acknowledge/cancel all alarms *)
   AUTO_FWD : BOOL ;                    (* Automated forward command *)
   MAN_FWD : BOOL ;                     (* Manual forward command *)
   MAN_FXWD_CHK : BOOL ;                (* Negated MAN_FWD for debouncing *)
   T_FWD_MAX : TIME ;                   (* Maximum time from FWD_CMD to FWD_FDBK *)
   FWD_FDBK : BOOL ;                    (* Confirmation of FWD_CMD completion *)
                                        (* by operative unit *)
   AUTO_REV : BOOL ;                    (* Automated reverse command *)
   MAN_REV : BOOL ;                     (* Manual reverse command *)
   MAN_REV_CHK : BOOL ;                 (* Negated MAN_REV for debouncing *)
   T_REV_MAX : TIME ;                   (* Maximum time from REV_CMD to REV_FDBK *)
   REV_FDBK : BOOL ;                    (* Confirmation of REV_CMD completion *)
END_VAR                                 (* by operative unit *)

VAR_OUTPUT KLAXON : BOOL ;              (* Any alarm active *)
   FWD_REV_ALRM : BOOL ;                (* Forward/reverse command conflict *)
   FWD_CMD : BOOL ;                     (* "Forward" command to operative unit *)
   FWD_ALRM : BOOL ;                    (* T_FWD_MAX expired without FWD_FDBK *)
   REV_CMD : BOOL ;                     (* "Reverse" command to operative unit *)
   REV_ALRM : BOOL ;                    (* T_REV_MAX expired without REV_FDBK *)
END_VAR

VAR FWD_MON : CMD_MONITOR ;             (* "Forward" command monitor *)
   REV_MON : CMD_MONITOR ;              (* "Reverse" command monitor *)
   FWD_REV_FF : SR ;                    (* Forward/reverse contention latch *)
END_VAR

(* Function block body *)
END_FUNCTION_BLOCK
```

The body of function block FWD_REV_MON can be written in the ST language as:

```
(* Evaluate internal function blocks *)
FWD_MON          (AUTO_MODE := AUTO,
                  ACK := ACK,
                  AUTO_CMD := AUTO_FWD,
                  MAN_CMD := MAN_FWD,
                  MAN_CMD_CHK := MAN_FWD_CHK,
                  T_CMD_MAX := T_FWD_MAX,
                  FDBK := FWD_FDBK) ;
REV_MON          (AUTO MODE := AUTO,
                  ACK := ACK,
                  AUTO_CMD := AUTO_REV,
                  MAN_CMD := MAN_REV,
                  MAN_CMD_CHK := MAN_REV_CHK,
                  T_CMD_MAX := T_REV_MAX,
                  FDBK := REV_FDBK) ;
FWD_REV_FF       (S1 := FWD_MON.CMD & REV_MON.CMD, R := ACK) ;

(* Transfer data to outputs *)

FWD_REV_ ALRM := FWD_REV_FF.Q1 ;
FWD_CMD := FWD_MON.CMD & NOT FWD_REV_ALRM ;
FWD_ ALRM := FWD_MON.ALRM ;
REV_CMD := REV_MON.CMD & NOT FWD_REV_ALRM ;
REV_ALRM := REV_MON.ALRM ;
KLAXON := FWD_ALRM OR REV_ALRM OR FWD_REV_ALRM ;
```

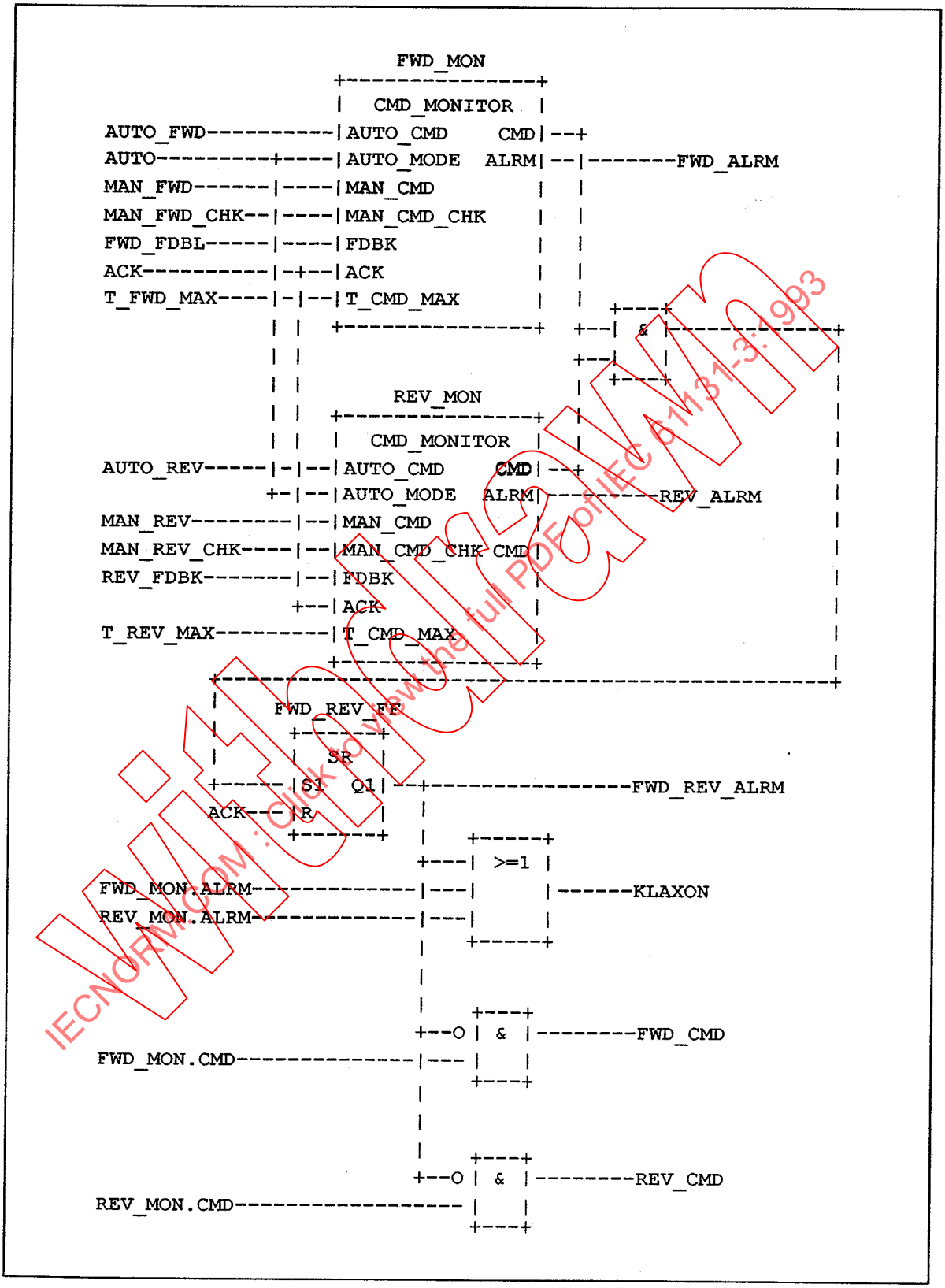
Le corps du bloc fonctionnel FWD_REV_MON dans le langage IL est le suivant:

```
LD      AUTO                    (* Load common inputs *)
ST      FWD_MON.AUTO_MODE
ST      REV_MON.AUTO_MODE
LD      ACK
ST      FWD_MON.ACK
ST      REV_MON.ACK
ST      FWD_REV_FF.R
LD      AUTO_FWD                (* Load inputs to FWD_MON *)
ST      FWD_MON.AUTO_CMD
LD      MAN_FWD
ST      FWD_MON.MAN_CMD
LD      MAN_FWD_CHK
ST      FWD_MON.MAN_CMD_CHK
LD      T_FWD_MAX
ST      FWD_MON.T_CMD_MAX
LD      FWD_FDBK
ST      FWD–MON.FDBK
CAL     FWD_MON                 (* Activate FWD_MON *)
LD      AUTO_REV                (* Load inputs to REV_MON *)
ST      REV_MON.AUTO_CMD
LD      MAN_REV
ST      REV_MON.MAN_CMD
LD      MAN_REV_CHK
ST      REV_MON.MAN_CMD_CHK
LD      T_REV_MAX
ST      REV_MON.T_CMD_MAX
LD      REV_FDBK
ST      REV_MON.FDBK
CAL     REV_MON                 (* Activate REV_MON *)
LD      FWD_MON.CMD             (* Check for contention *)
AND     REV_MON.CMD
S1      FWD_REV_FF              (* Latch contention condition *)
LD      FWD_REV_FF.Q
ST      FWD_REV_ALRM            (* Contention alarm *)
LD      FWD_MON.CMD             (* "Forward" command and alarm *)
ANDN    FWD_REV_ALRM
ST      FWD_CMD
LD      FWD_MON.ALRM
ST      FWD_ALRM
LD      REV_MON.CMD             (* "Reverse" command and alarm *)
ANDN    FWD_REV_ALRM
ST      REV_CMD
LD      REV_MON.ALRM
ST      REV_ALRM
OR      FWD_ALRM                (* OR all alarms *)
OR      FWD_REV_ALRM
ST      KLAXON
```

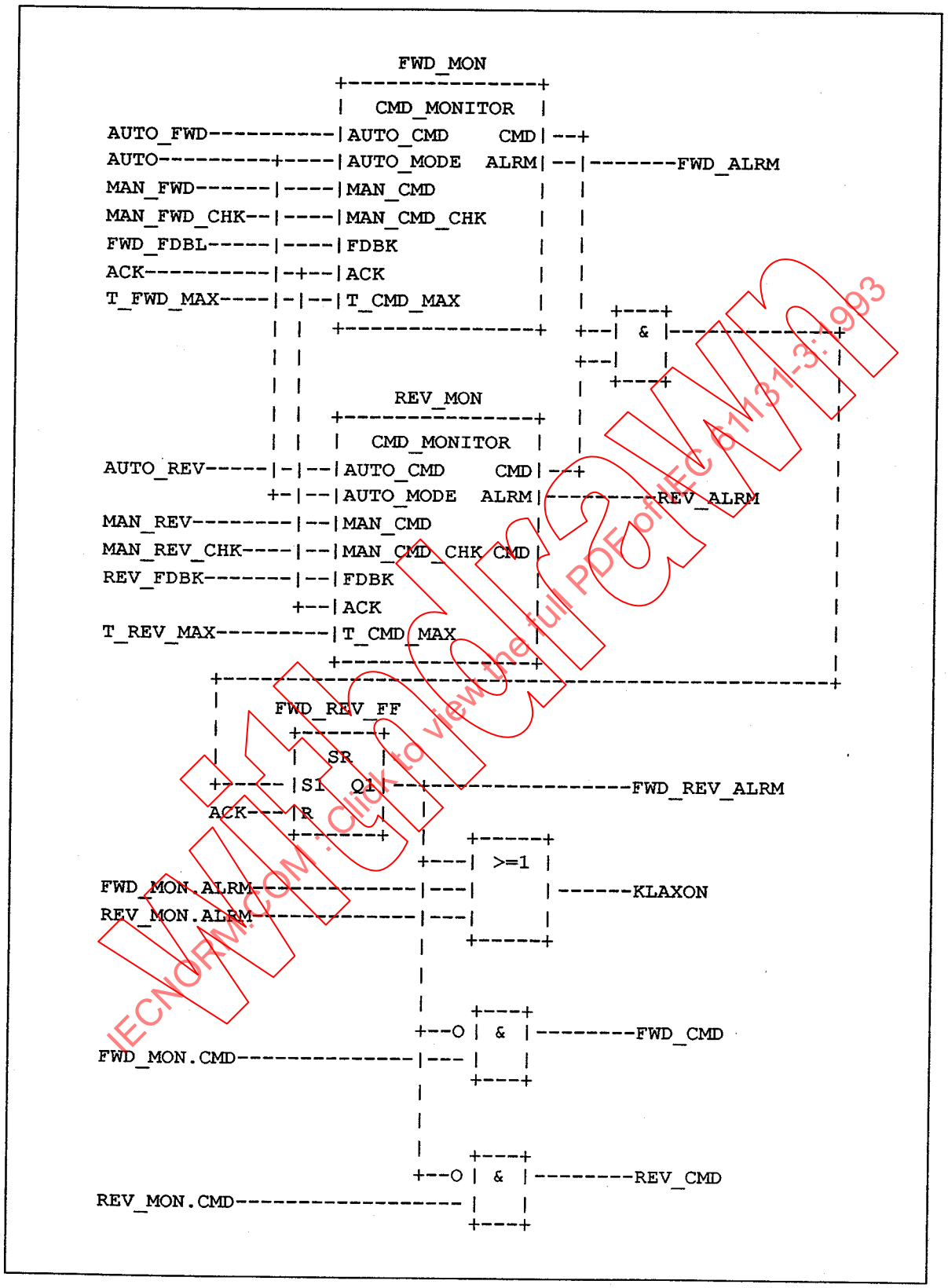
The body of function block FWD_REV_MON in the IL language is:

```
LD      AUTO                    (* Load common inputs *)
ST      FWD_MON.AUTO_MODE
ST      REV_MON.AUTO_MODE
LD      ACK
ST      FWD_MON.ACK
ST      REV_MON.ACK
ST      FWD_REV_FF.R
LD      AUTO_FWD                (* Load inputs to FWD_MON *)
ST      FWD_MON.AUTO_CMD
LD      MAN_FWD
ST      FWD_MON.MAN_CMD
LD      MAN_FWD_CHK
ST      FWD_MON.MAN_CMD_CHK
LD      T_FWD_MAX
ST      FWD_MON.T_CMD_MAX
LD      FWD_FDBK
ST      FWD_MON.FDBK
CAL     FWD_MON                 (* Activate FWD_MON *)
LD      AUTO_REV                (* Load inputs to REV_MON *)
ST      REV_MON.AUTO_CMD
LD      MAN_REV
ST      REV_MON.MAN_CMD
LD      MAN_REV_CHK
ST      REV_MON.MAN_CMD_CHK
LD      T_REV_MAX
ST      REV_MON.T_CMD_MAX
LD      REV_FDBK
ST      REV_MON.FDBK
CAL     REV_MON                 (* Activate REV_MON *)
LD      FWD_MON.CMD             (* Check for contention *)
AND     REV_MON.CMD
S1      FWD_REV_FF              (* Latch contention condition *)
LD      FWD_REV_FF.Q
ST      FWD_REV_ALRM            (* Contention alarm *)
LD      FWD_MON.CMD             (* "Forward" command and alarm *)
ANDN    FWD_REV_ALRM
ST      FWD_CMD
LD      FWD_MON.ALRM
ST      FWD_ALRM
LD      REV_MON.CMD             (* "Reverse" command and alarm *)
ANDN    FWD_REV_ALRM
ST      REV_CMD
LD      REV_MON.ALRM
ST      REV_ALRM
OR      FWD_ALRM                (* OR all alarms *)
OR      FWD_REV_ALRM
ST      KLAXON
```

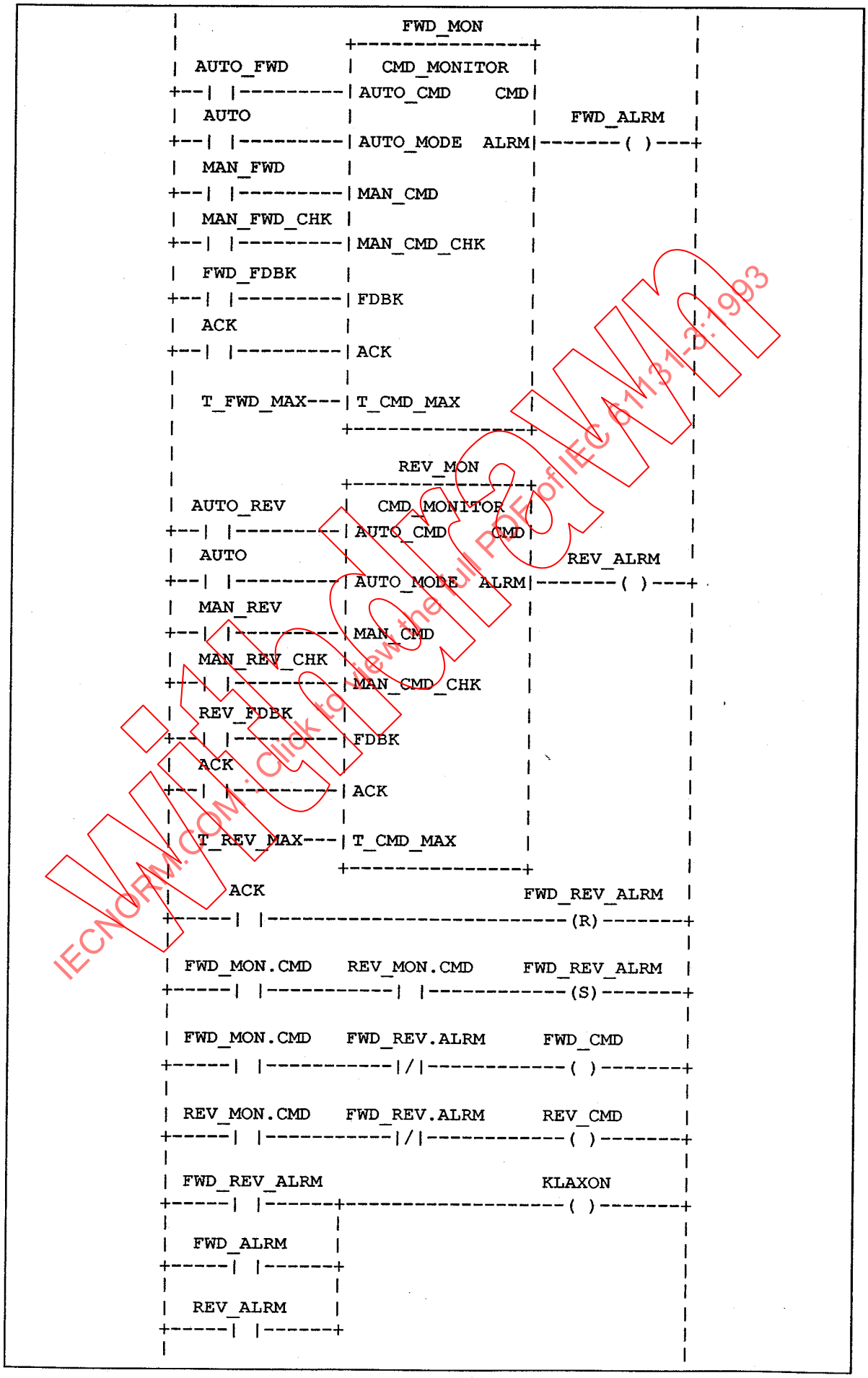
Le corps du bloc fonctionnel FWD_REV_MON dans le langage FBD est le suivant:



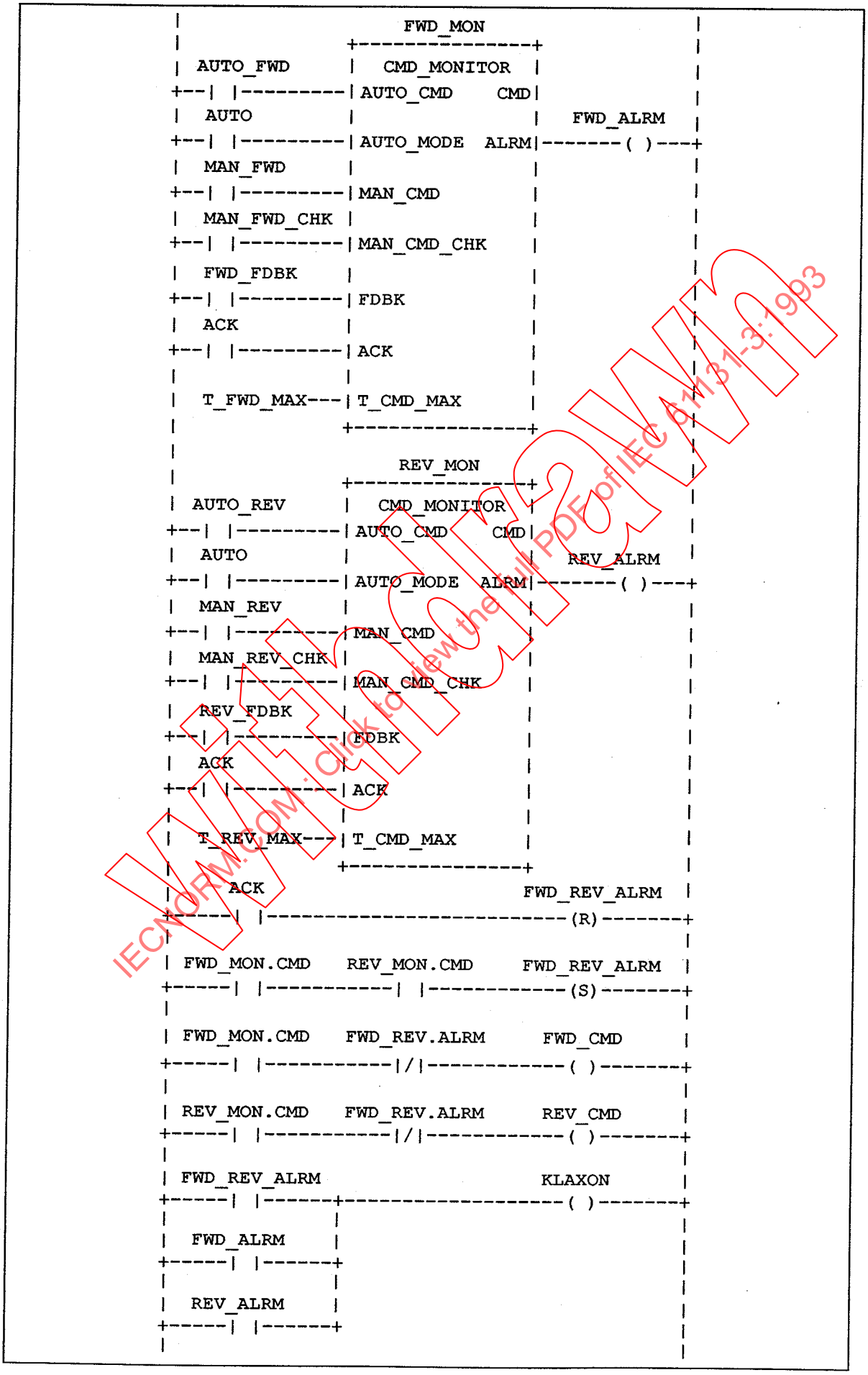
The body of function block FWD_REV_MON in the FBD language is:



Le corps du bloc fonctionnel FWD_REV_MON dans le langage LD est le suivant:



The body of function block FWD_REV_MON in the LD language is:



F.4 Bloc fonctionnel STACK_INT

Ce bloc fonctionnel permet d'empiler jusqu'à 128 entiers. Les opérations habituelles d'empilage PUSH et POP sont assurées par des entrées Booléennes déclenchées par des fronts. Une commande prédominante de remise à l'état est assurée; l'épaisseur maximale de pile (N) est déterminée au moment de la remise à l'état. En plus de la donnée du sommet de pile, des sorties booléennes sont prévues indiquant que la pile est vide et que la pile déborde.

Une forme littérale de la déclaration de ce bloc fonctionnel est la suivante:

```
FUNCTION_BLOCK STACK_INT
  VAR_INPUT PUSH, POP : BOOL R_EDGE ;        (* Basic stack operations *)
            R1 : BOOL ;                      (* Over-riding reset *)
            IN : INT ;                       (* Input to be pushed *)
            N : INT ;                        (* Maximum depth after reset *)
  END_VAR
  VAR_OUTPUT EMPTY : BOOL := 1 ;             (* Stack empty *)
             OFLO : BOOL := 0 ;              (* Stack overflow *)
             OUT : INT := 0 ;                (* Top of stack data *)
  END_VAR
  VAR STK : ARRAY[0..127] OF INT ;           (* Internal stack *)
            NI : INT := 128  ;               (* Storage for N upon reset *)
            PTR : INT := -1 ;                (* Stack pointer *)
  END_VAR

  (* Function block body *)

  END_FUNCTION_BLOCK
```

Une déclaration graphique du bloc fonctionnel STACK_INT est la suivante:

```
                    +-----------+
                    | STACK_INT |
BOOL--->PUSH   EMPTY|---BOOL
BOOL--->POP     OFLO|---BOOL
BOOL---|R1       OUT|---INT
INT----|IN          |
INT----|N           |
                    +-----------+

(* Internal variable declarations *)

VAR STK : ARRAY[0..127] OF INT ;              (* Internal stack *)
    NI : INT := 128 ;                         (* Storage for N upon reset *)
    PTR : INT := -1 ;                         (* Stack pointer *)
END_VAR
```

F.4 Function block STACK_INT

This function block provides a stack of up to 128 integers. The usual stack operations of PUSH and POP are provided by edge-triggered Boolean inputs. An overriding reset (R1) input is provided; the maximum stack depth (N) is determined at the time of resetting. In addition to the top-of-stack data (OUT), Boolean outputs are provided indicating stack empty and stack overflow states.

A textual form of the declaration of this function block is:

```
FUNCTION_BLOCK STACK_INT
  VAR_INPUT PUSH, POP : BOOL R_EDGE ;        (* Basic stack operations *)
            R1 : BOOL ;                      (* Over-riding reset *)
            IN : INT ;                       (* Input to be pushed *)
            N : INT ;                        (* Maximum depth after reset *)
  END_VAR
  VAR_OUTPUT EMPTY : BOOL := 1 ;             (* Stack empty *)
            OFLO : BOOL := 0 ;               (* Stack overflow *)
            OUT : INT := 0 ;                 (* Top of stack data *)
  END_VAR
  VAR STK : ARRAY[0..127] OF INT ;           (* Internal stack *)
            NI : INT := 128  ;               (* Storage for N upon reset *)
            PTR : INT := -1 ;                (* Stack pointer *)
  END_VAR

  (* Function block body *)

  END_FUNCTION_BLOCK
```

A graphical declaration of function block STACK_INT is:

```
                  +------------+
                  | STACK_INT  |
          BOOL--->PUSH   EMPTY|---BOOL
          BOOL--->POP     OFLO|---BOOL
          BOOL---|R1        OUT|---INT
          INT----|IN           |
          INT----|N            |
                 +------------+

(* Internal variable declarations *)

VAR STK : ARRAY[0..127] OF INT ;              (* Internal stack *)
    NI : INT := 128 ;                         (* Storage for N upon reset *)
    PTR : INT := -1 ;                         (* Stack pointer *)
END_VAR
```

Le corps du bloc fonctionnel dans le langage ST est le suivant:

```
IF R1 THEN
  OFLO := 0 ; EMPTY := 1 ; PTR := -1 ;
  NI := LIMIT (MN := 1,IN := N,MX := 128) ; OUT := 0 ;
ELSIF POP & NOT EMPTY THEN
  OFLO := 0 ; PTR := PTR-1 ; EMPTY := PTR < 0 ;
  IF EMPTY THEN OUT := 0 ;
  ELSE OUT := STK[PTR] ;
  END_IF ;
ELSIF PUSH & NOT OFLO THEN
  EMPTY := 0 ; PTR := PTR+1 ; OFLO := (PTR = NI) ;
  IF NOT OFLO THEN OUT := IN ; STK[PTR] := IN ;
  ELSE OUT := 0 ;
  END_IF ;
END_IF ;
```