

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**Function blocks –
Part 1: Architecture**

**Blocs fonctionnels –
Partie 1: Architecture**



THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2012 IEC, Geneva, Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either IEC or IEC's member National Committee in the country of the requester.

If you have any questions about IEC copyright or have an enquiry about obtaining additional rights to this publication, please contact the address below or your local IEC member National Committee for further information.

Droits de reproduction réservés. Sauf indication contraire, aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de la CEI ou du Comité national de la CEI du pays du demandeur.

Si vous avez des questions sur le copyright de la CEI ou si vous désirez obtenir des droits supplémentaires sur cette publication, utilisez les coordonnées ci-après ou contactez le Comité national de la CEI de votre pays de résidence.

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

Useful links:

IEC publications search - www.iec.ch/searchpub

The advanced search enables you to find IEC publications by a variety of criteria (reference number, text, technical committee,...).

It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available on-line and also once a month by email.

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary (IEV) on-line.

Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

A propos de la CEI

La Commission Electrotechnique Internationale (CEI) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications CEI

Le contenu technique des publications de la CEI est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Liens utiles:

Recherche de publications CEI - www.iec.ch/searchpub

La recherche avancée vous permet de trouver des publications CEI en utilisant différents critères (numéro de référence, texte, comité d'études,...).

Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

Just Published CEI - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications de la CEI. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org

Le premier dictionnaire en ligne au monde de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans les langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (VEI) en ligne.

Service Clients - webstore.iec.ch/csc

Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: csc@iec.ch.

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**Function blocks –
Part 1: Architecture**

**Blocs fonctionnels –
Partie 1: Architecture**

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

COMMISSION
ELECTROTECHNIQUE
INTERNATIONALE

PRICE CODE
CODE PRIX

XF

ICS 25.040; 35.240.50

ISBN 978-2-83220-481-8

**Warning! Make sure that you obtained this publication from an authorized distributor.
Attention! Veuillez vous assurer que vous avez obtenu cette publication via un distributeur agréé.**

CONTENTS

FOREWORD.....	5
INTRODUCTION.....	7
1 Scope.....	8
2 Normative references	8
3 Terms and definitions	9
4 Reference models	18
4.1 System model.....	18
4.2 Device model	19
4.3 Resource model	19
4.4 Application model.....	21
4.5 Function block model.....	21
4.5.1 Characteristics of function block instances	21
4.5.2 Function block type specifications	23
4.5.3 Execution model for basic function blocks	23
4.6 Distribution model	25
4.7 Management model	25
4.8 Operational state models.....	27
5 Specification of function block, subapplication and adapter interface types.....	27
5.1 Overview	27
5.2 Basic function blocks.....	28
5.2.1 Type declaration	28
5.2.2 Behavior of instances	30
5.3 Composite function blocks.....	33
5.3.1 Type specification.....	33
5.3.2 Behavior of instances	35
5.4 Subapplications	36
5.4.1 Type specification.....	36
5.4.2 Behavior of instances	37
5.5 Adapter interfaces	38
5.5.1 General principles	38
5.5.2 Type specification.....	38
5.5.3 Usage.....	39
5.6 Exception and fault handling.....	41
6 Service interface function blocks	41
6.1 General principles	41
6.1.1 General	41
6.1.2 Type specification.....	42
6.1.3 Behavior of instances	43
6.2 Communication function blocks	45
6.2.1 Type specification.....	45
6.2.2 Behavior of instances	46
6.3 Management function blocks	47
6.3.1 Requirements	47
6.3.2 Type specification.....	47
6.3.3 Behavior of managed function blocks.....	50
7 Configuration of functional units and systems	52

7.1	Principles of configuration	52
7.2	Functional specification of resource, device and segment types	52
7.2.1	Functional specification of resource types	52
7.2.2	Functional specification of device types	53
7.2.3	Functional specification of segment types	53
7.3	Configuration requirements	53
7.3.1	Configuration of systems	53
7.3.2	Specification of applications	54
7.3.3	Configuration of devices and resources	54
7.3.4	Configuration of network segments and links	55
Annex A (normative)	Event function blocks	56
Annex B (normative)	Textual syntax	63
Annex C (informative)	Object models	74
Annex D (informative)	Relationship to IEC 61131-3	82
Annex E (informative)	Information exchange	92
Annex F (normative)	Textual specifications	100
Annex G (informative)	Attributes	113
Bibliography		117
Figure 1 – System model		18
Figure 2 – Device model		19
Figure 3 – Resource model		20
Figure 4 – Application model		21
Figure 5 – Characteristics of function blocks		22
Figure 6 – Execution model		24
Figure 7 – Execution timing		24
Figure 8 – Distribution and management models		26
Figure 9 – Function block and subapplication types		28
Figure 10 – Basic function block type declaration		29
Figure 11 – ECC example		30
Figure 12 – ECC operation state machine		32
Figure 13 – Composite function block <code>PI_REAL</code> example		34
Figure 14 – Basic function block <code>PID_CALC</code> example		35
Figure 15 – Subapplication <code>PI_REAL_APPL</code> example		37
Figure 16 – Adapter interfaces – Conceptual model		38
Figure 17 – Adapter type declaration – graphical example		39
Figure 18 – Illustration of provider and acceptor function block type declarations		40
Figure 19 – Illustration of adapter connections		41
Figure 20 – Example service interface function blocks		43
Figure 21 – Example service sequence diagrams		44
Figure 22 – Generic management function block type		47
Figure 23 – Service primitive sequences for unsuccessful service		48
Figure 24 – Operational state machine of a managed function block		51
Figure A.1 – Event split and merge		62

Figure C.1 – ESS overview	74
Figure C.2 – Library elements	75
Figure C.3 – Declarations	76
Figure C.4 – Function block network declarations	77
Figure C.5 – Function block type declarations	79
Figure C.6 – IPMCS overview	79
Figure C.7 – Function block types and instances	81
Figure D.1 – Example of a “simple” function block type	82
Figure D.2 – Function block type READ	85
Figure D.3 – Function block type UREAD	87
Figure D.4 – Function block type WRITE	88
Figure D.5 – Function block type TASK	90
Figure E.1 – Type specifications for unidirectional transactions	93
Figure E.2 – Connection establishment for unidirectional transactions	93
Figure E.3 – Normal unidirectional data transfer	93
Figure E.4 – Connection release in unidirectional data transfer	94
Figure E.5 – Type specifications for bidirectional transactions	94
Figure E.6 – Connection establishment for bidirectional transaction	95
Figure E.7 – Bidirectional data transfer	95
Figure E.8 – Connection release in bidirectional data transfer	95
Table 1 – States and transitions of ECC operation state machine	32
Table 2 – Standard inputs and outputs for service interface function blocks	42
Table 3 – Service primitive semantics	45
Table 4 – Variable semantics for communication function blocks	46
Table 5 – Service primitive semantics for communication function blocks	46
Table 6 – CMD input values and semantics	48
Table 7 – STATUS output values and semantics	48
Table 8 – Command syntax	49
Table 9 – Semantics of actions in Figure 24	52
Table A.1 – Event function blocks	57
Table C.1 – ESS class descriptions	75
Table C.2 – Syntactic productions for library elements	75
Table C.3 – Syntactic productions for declarations	77
Table C.4 – IPMCS classes	80
Table D.1 – Semantics of STATUS values	83
Table D.2 – Source code of function block type READ	86
Table D.3 – Source code of function block type UREAD	87
Table D.4 – Source code of function block type WRITE	89
Table D.5 – Source code of function block type TASK	90
Table D.6 – IEC 61499 interoperability features	91
Table E.1 – COMPACT encoding of fixed length data types	99
Table G.1 – Elements of attribute definitions	114

INTERNATIONAL ELECTROTECHNICAL COMMISSION

FUNCTION BLOCKS –

Part 1: Architecture

FOREWORD

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

International Standard IEC 61499-1 has been prepared by subcommittee 65B: Measurement and control devices, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2005. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

- *Execution control* in basic function blocks (5.2) has been clarified and extended:
 - Dynamic and static parts of the EC transition condition are clearly delineated by using the `ec_transition_event[guard_condition]` syntax of the Unified Modeling Language (UML) (5.2.1.3, B.2.1).
 - The terminology "crossing of an EC transition" (3.10) is used preferentially to "clearing" to avoid the misinterpretation that the entire transition condition corresponds to a Boolean variable that can be "cleared."

- Operation of the ECC state machine in 5.2.2.2 has been clarified and made more rigorous.
- Event and data outputs of adapter instances (plugs and sockets) can be used in EC transition conditions, and event inputs of adapter instances can be used as EC action outputs.
- *Temporary variables* (3.97) can be declared (B.2.1) and used in algorithms of basic function blocks.
- *Service sequences* (6.1.3) can now be defined for basic and composite function block types and adapter types, as well as service interface types.
- The syntax for *mapping* of FB instances from applications to resources has been simplified (Clause B.3).
- Syntax for definition of *segment types* (7.2.3) for network segments of system configurations has been added (Clause B.3).
- Function block types for interoperation with programmable controllers are defined (Clause D.6).
- The READ/WRITE management commands (Table 8) now apply only to *parameters*.

The text of this part of IEC 61499 is based on the following documents:

FDIS	Report on voting
65B/845/FDIS	65B/855/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table (when voting is completed).

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 61499 series can be found, under the general title *Function blocks*, on the IEC website.

Terms used throughout this International Standard that have been defined in Clause 3 appear in *italics*.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication will be

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

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

IEC 61499 consists of the following parts, under the general title *Function blocks*:

- Part 1 (this document) contains:
 - general requirements, including scope, normative references, definitions, and reference models;
 - rules for the declaration of *function block types*, and rules for the behavior of *instances* of the types so declared;
 - rules for the use of function blocks in the *configuration* of distributed industrial-process measurement and control *systems* (IPMCSs);
 - rules for the use of function blocks in meeting the communication requirements of distributed IPMCSs;
 - rules for the use of function blocks in the management of *applications*, *resources* and *devices* in distributed IPMCSs.
- Part 2 defines requirements for *software tools* to support the following systems engineering tasks:
 - the specification of *function block types*;
 - the functional specification of *resource types* and *device types*;
 - the specification, analysis, and validation of distributed IPMCSs;
 - the *configuration*, *implementation*, operation, and maintenance of distributed IPMCSs;
 - the exchange of *information* among *software tools*.
- Part 3 (Tutorial information) has been withdrawn due to the widespread current availability of tutorial and educational materials regarding IEC 61499. However, an updated 2nd Edition of Part 3 may be developed in the future.
- Part 4 defines rules for the development of *compliance profiles* which specify the features of IEC 61499-1 and IEC 61499-2 to be implemented in order to promote the following attributes of IEC 61499-based systems, devices and software tools:
 - interoperability of devices from multiple suppliers;
 - portability of software between software tools of multiple suppliers; and
 - configurability of devices from multiple vendors by software tools of multiple suppliers.

FUNCTION BLOCKS –

Part 1: Architecture

1 Scope

This part of IEC 61499 defines a generic architecture and presents guidelines for the use of *function blocks* in distributed industrial-process measurement and control systems (IPMCSs). This architecture is presented in terms of implementable reference *models*, textual syntax and graphical representations. These models, representations and syntax **can be used for**:

- the specification and standardization of *function block types*;
- the functional specification and standardization of system elements;
- the implementation independent specification, analysis, and validation of distributed IPMCSs;
- the *configuration, implementation, operation, and maintenance* of distributed IPMCSs;
- the exchange of *information* among *software tools* for the performance of the above *functions*.

This part of IEC 61499 does not restrict or specify the functional capabilities of IPMCSs or their system elements, except as such capabilities are represented using the elements defined herein. IEC 61499-4 addresses the extent to which the elements defined in this standard may be restricted by the functional capabilities of compliant systems, subsystems, and devices.

Part of the purpose of this standard is to provide reference models for the use of function blocks in other standards dealing with the support of the system life cycle, including system planning, design, implementation, validation, operation and maintenance. The models given in this standard are intended to be generic, domain independent and extensible to the definition and use of function blocks in other standards or for particular applications or application domains. It is intended that specifications written according to the rules given in this standard be concise, implementable, complete, unambiguous, and consistent.

NOTE 1 The provisions of this standard alone are not sufficient to ensure interoperability among devices of different vendors. Standards complying with this part of IEC 61499 can specify additional provisions to ensure such interoperability.

NOTE 2 Standards complying with this part of IEC 61499 can specify additional provisions to enable the performance of *system, device, resource* and *application* management *functions*.

2 Normative references

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

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

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

IEC/ISO 7498-1:1994, *Information technology – Open systems interconnection – Basic reference model: The basic model*

ISO/IEC 8824-1:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*

ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

NOTE Terms defined in Clause 3 are *italicized* where they appear in definitions and Notes to entry of other terms as well as throughout the body of the document.

3.1

acceptor

function block instance which provides a *socket adapter* of a defined *adapter interface type*

3.2

adapter connection

connection from a *plug adapter* to a *socket adapter* of the same *adapter interface type*, which carries the flows of *data* and *events* defined by the *adapter interface type*

3.3

adapter interface type

type which consists of the definition of a set of *event inputs*, *event outputs*, *data inputs*, and *data outputs*, and whose *instances* are *plug adapters* and *socket adapters*

3.4

algorithm

finite set of well-defined rules for the solution of a problem in a finite number of *operations*

3.5

application

software functional unit that is specific to the solution of a problem in industrial-process measurement and control

Note 1 to entry: An application can be distributed among *resources*, and might communicate with other applications.

3.6

attribute

property or characteristic of an *entity*, for instance, the version identifier of a *function block type* specification

3.7

basic function block type

function block type that cannot be decomposed into other function blocks and that utilizes an *execution control chart (ECC)* to control the *execution* of its *algorithms*

3.8

bidirectional transaction

transaction in which a request and possibly *data* are conveyed from an *requester* to a *responder*, and in which a response and possibly *data* are conveyed from the responder back to the requester

3.9

character

member of a set of elements that is used for the representation, organization, or control of *data*

3.10

crossing

clearing

<of an EC transition> *operation* by means of which control is passed from the predecessor *EC state* of an *EC transition* to its successor EC state

Note 1 to entry: This operation consists of de-activation of the predecessor EC state, followed by activation of the successor EC state.

3.11

communication connection

connection that utilizes the communication mapping function of one or more *resources* for the conveyance of *information*

3.12

communication function block

service interface function block that represents the *interface* between an *application* and the communication mapping function of a *resource*

3.13

communication function block type

function block type whose *instances* are *communication function blocks*

3.14

component function block

function block instance which is used in the specification of an *algorithm* of a *composite function block type*

Note 1 to entry: A component function block can be of *basic*, *composite* or *service interface type*.

3.15

component subapplication

subapplication instance that is used in the specification of a *subapplication type*

3.16

composite function block type

function block type whose *algorithms* and the control of their *execution* are expressed entirely in terms of interconnected *component function blocks*, *events*, and *variables*

3.17

concurrent

pertaining to *algorithms* that are *executed* during a common period of time during which they may have to alternately share common *resources*

3.18

configuration (of a system or device)

selecting *functional units*, assigning their locations and defining their interconnections

3.19

configuration parameter

parameter related to the *configuration* of a *system*, *device* or *resource*

3.20**confirm primitive**

service primitive which represents an interaction in which a *resource* indicates completion of some *algorithm* previously *invoked* by an interaction represented by a *request primitive*

3.21**connection**

association established between *functional units* for conveying *information*

3.22**critical region**

operation or sequence of operations which is *executed* under the exclusive control of a locking object which is associated with the *data* on which the operations are performed

3.23**data**

reinterpretable representation of *information* in a formalized manner suitable for communication, interpretation or processing

3.24**data connection**

association between two *function blocks* for the conveyance of *data*

3.25**data input**

interface of a *function block* which receives *data* from a *data connection*

3.26**data output**

interface of a *function block* which supplies *data* to a *data connection*

3.27**data type**

set of values together with a set of permitted *operations*

3.28**declaration**

mechanism for establishing the definition of an *entity*

Note 1 to entry: A declaration can involve attaching an *identifier* to the entity, and allocating *attributes* such as *data types* and *algorithms* to it.

3.29**device**

independent physical *entity* capable of performing one or more specified *functions* in a particular context and delimited by its *interfaces*

Note 1 to entry: A *programmable controller system* as defined in IEC 61131-1 is a *device*.

3.30**device management application**

application whose primary function is the management of multiple *resources* within a *device*

3.31**entity**

particular thing, such as a person, place, *process*, object, concept, association, or *event*

3.32

event

instantaneous occurrence that is significant to scheduling the *execution* of an *algorithm*

Note 1 to entry: The execution of an algorithm may make use of *variables* associated with an event.

3.33

event connection

association among *function blocks* for the conveyance of *events*

3.34

event input

interface of a *function block* which can receive *events* from an *event connection*

3.35

event output

interface of a *function block* which can issue *events* to an *event connection*

3.36

exception

event that causes suspension of normal *execution*

3.37

execution

process of carrying out a sequence of *operations* specified by an *algorithm*

Note 1 to entry: The sequence of operations to be executed may vary from one *invocation* of a *function block instance* to another, depending on the rules specified by the function block's *algorithm* and the current values of *variables* in the function block's data structure.

3.38

execution control action

EC action

element associated with an *execution control state*, which identifies an *algorithm* to be *executed*, an *event* to be issued, or both

Note 1 to entry: Timing of algorithm execution and event issuance are addressed in 5.2.2.

3.39

execution control chart

ECC

graphical or textual representation of the causal relationships among *events* at the *event inputs* and *event outputs* of a *function block* and the *execution* of the function block's *algorithms*, using *execution control states*, *execution control transitions*, and *execution control actions*

3.40

execution control initial state

EC initial state

execution control state that is active upon initialization of an *execution control chart*

3.41

execution control state

EC state

situation in which the behavior of a *basic function block* with respect to its *variables* is determined by the *algorithms* associated with a specified set of *execution control actions*

3.42**execution control transition****EC transition**

means by which control passes from a predecessor *execution control state* to a successor *execution control state*

3.43**fault**

abnormal condition that may cause a reduction in, or loss of, the capability of a *functional unit* to perform a required *function*

3.44**function**

specific purpose of an *entity* or its characteristic action

3.45**function block****function block instance**

software functional unit comprising an individual, named copy of a data structure upon which associated *operations* may be performed as specified by a corresponding *function block type*

Note 1 to entry: Typical operations of a function block include modification of the values of the data in its associated data structure.

Note 2 to entry: The *function block instance* and its corresponding *function block type* defined in IEC 61131-3 are programming language elements with a different set of features.

3.46**function block network**

network whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*

Note 1 to entry: This is a generalization of the *function block diagram* defined in IEC 61131-3.

3.47**function block type**

type whose *instances* are *function blocks*

Note 1 to entry: Function block types include basic function block types, composite function block types, and service interface function block types

3.48**functional unit**

entity of *hardware* or *software*, or both, capable of accomplishing a specified purpose

3.49**hardware**

physical equipment, as opposed to programs, procedures, rules and associated documentation

3.50**identifier**

one or more *characters* used to name an *entity*

3.51**implementation**

development phase in which the *hardware* and *software* of a *system* become operational

3.52

indication primitive

service primitive which represents an interaction in which a *resource* either

- a) indicates that it has, on its own initiative, *invoked* some *algorithm*; or
- b) indicates that an *algorithm* has been invoked by a peer *application*

3.53

information

meaning that is currently assigned to *data* by means of the conventions applied to that data

3.54

input variable

variable whose value is supplied by a *data input*, and which may be used in one or more *operations* of a *function block*

Note 1 to entry: An *input parameter* of a *function block*, as defined in IEC 61131-3, is an *input variable*.

3.55

instance

functional unit comprising an individual, named *entity* with the *attributes* of a defined *type*

3.56

instance name

identifier associated with and designating an *instance*

3.57

instantiation

creation of an *instance* of a specified *type*

3.58

interface

shared boundary between two *functional units*, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate

3.59

internal operation

<of a *function block*> *operation* associated with an *algorithm* of a function block, with its *execution* control, or with the functional capabilities of the associated *resource*

3.60

internal variable

variable whose value is used or modified by one or more *operations* of a *function block*, but is not supplied by a *data input* or to a *data output*

3.61

invocation

process of initiating the *execution* of the sequence of *operations* specified in an *algorithm*

3.62

link

design element describing the *connection* between a *device* and a *network segment*

3.63

literal

lexical unit that directly represents a value

3.64**management function block**

function block whose primary *function* is the management of *applications* within a *resource*

3.65**management resource**

resource whose primary *function* is the management of other *resources*

3.66**mapping**

set of features or *attributes* having defined correspondence with the members of another set

3.67**message**

ordered series of *characters* intended to convey *information*

3.68**message sink**

part of a communication *system* in which *messages* are considered to be received

3.69**message source**

part of a communication *system* from which *messages* are considered to originate

3.70**model**

mathematical or physical representation of a system or a process

3.71**multitasking**

mode of operation that provides for the *concurrent execution* of two or more *algorithms*

3.72**network**

arrangement of nodes and interconnecting branches

3.73**operation**

well-defined action that, when applied to any permissible combination of known *entities*, produces a new *entity*

3.74**output variable**

variable whose value is established by one or more *operations* of a *function block*, and is supplied to a *data output*

Note 1 to entry: An *output parameter* of a *function block*, as defined in IEC 61131-3, is an *output variable*.

3.75**parameter**

variable that is given a constant value for a specified *application* and that may denote the application

3.76**plug****plug adapter**

instance of an *adapter interface type* which provides a starting point for an *adapter connection* from a *provider function block*

3.77

provider

function block instance which provides a *plug adapter* of a defined *adapter interface type*

3.78

request primitive

service primitive which represents an interaction in which an *application* invokes some *algorithm* provided by a *service*

3.79

requester

functional unit which initiates a *transaction* via a *request primitive*

3.80

resource

functional unit which has independent control of its operation, and which provides various *services* to *applications*, including the scheduling and *execution* of *algorithms*

Note 1 to entry: The RESOURCE defined in IEC 61131-3:2003, 1.3.66 is a programming language element corresponding to the *resource* defined above.

Note 2 to entry: A *device* contains one or more resources.

3.81

resource management application

application whose primary function is the management of a single *resource*

3.82

responder

functional unit which concludes a *transaction* via a *response primitive*

3.83

response primitive

service primitive which represents an interaction in which an *application* indicates that it has completed some *algorithm* previously *invoked* by an interaction represented by an *indication primitive*

3.84

sample, verb

to sense and retain the instantaneous value of a *variable* for later use

3.85

scheduling function

function which selects *algorithms* or *operations* for *execution*, and initiates and terminates such execution

3.86

segment

physical partition of a *communication network*

3.87

service

functional capability of a *resource* which can be modeled by a sequence of *service primitives*

3.88

service interface function block

function block which provides one or more *services* to an *application*, based on a *mapping* of *service primitives* to the function block's *event inputs*, *event outputs*, *data inputs* and *data outputs*

3.89**service primitive**

abstract, implementation-independent representation of an interaction between an *application* and a *resource*

3.90**service sequence diagram**

diagram representing a sequence of *service primitives*

3.91**socket****socket adapter**

instance of an *adapter interface type* which provides an end point for an *adapter connection* to an *acceptor* function block

3.92**software**

intellectual creation comprising the programs, procedures, rules, *configurations* and any associated documentation pertaining to the operation of a *system*

3.93**software tool**

software that is used for the production, inspection or analysis of other software

3.94**subapplication instance**

instance of a *subapplication type* inside an *application* or inside a subapplication type

Note 1 to entry: A subapplication instance may be distributed among *resources*, i.e. its component function blocks or the content of its component subapplications may be assigned to different resources.

3.95**subapplication type**

functional unit whose body consists of interconnected *component function blocks* or *component subapplications*

Note 1 to entry: A subapplication type enables the creation of substructures of *applications* in the form of a self-similar hierarchy.

3.96**system**

set of interrelated elements considered in a defined context as a whole and separated from its environment

Note 1 to entry: Such elements may be both material objects and concepts as well as the results thereof (e.g. forms of organisation, mathematical methods, and programming languages).

Note 2 to entry: The system is considered to be separated from the environment and other external systems by an imaginary surface, which can cut the links between them and the considered system.

3.97**temporary variable**

variable whose value is initialized, used and possibly modified during *execution* of an *algorithm*; that is not visible outside the body of the algorithm, and whose value does not persist from one execution of the algorithm to the next

3.98**transaction**

unit of service in which a request and possibly *data* is conveyed from a *requester* to a *responder*, and in which a response and possibly data may also be conveyed from the responder back to the requester

3.99

type

software element which specifies the common *attributes* shared by all *instances* of the type

3.100

type name

identifier associated with and designating a *type*

3.101

unidirectional transaction

transaction in which a request and possibly *data* is/are conveyed from an *requester* to a *responder*, and in which a response is not conveyed from the responder back to the requester

3.102

variable

software entity that may take different values, one at a time

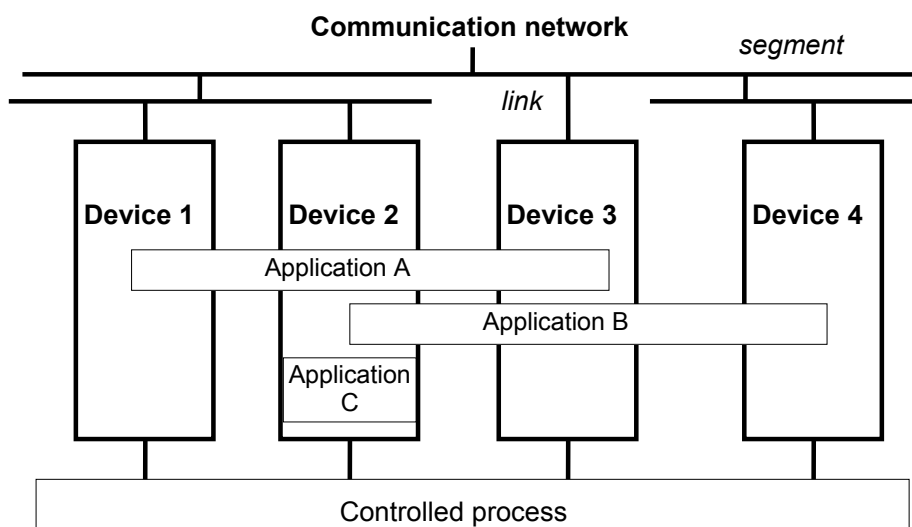
Note 1 to entry: The values of a variable are usually restricted to a certain *data type*.

Note 2 to entry: Variables may be classified as *input variables*, *output variables*, *internal variables* and *temporary variables*.

4 Reference models

4.1 System model

For the purposes of IEC 61499, an industrial process measurement and control *system* (IPMCS) is modeled, as shown in Figure 1, as a collection of *devices* interconnected and communicating with each other by means of a communication *network* consisting of *segments* and *links*. Devices are connected to network segments via *links*.



NOTE The controlled process is not part of the measurement and control system.

Figure 1 – System model

A *function* performed by the IPMCS is modeled as an *application* which may reside in a single device, such as application C in Figure 1, or may be distributed among several devices, such as applications A and B in Figure 1. For instance, an application may consist of one or more control loops in which the input sampling is performed in one device, control processing is performed in another, and output conversion in a third.

4.2 Device model

As illustrated in Figure 2, a *device* shall contain at least one *interface*, that is, process interface or communication interface, and can contain zero or more *resources*.

NOTE 1 A device is considered to be an *instance* of a corresponding device *type*, defined as specified in 7.2.2.

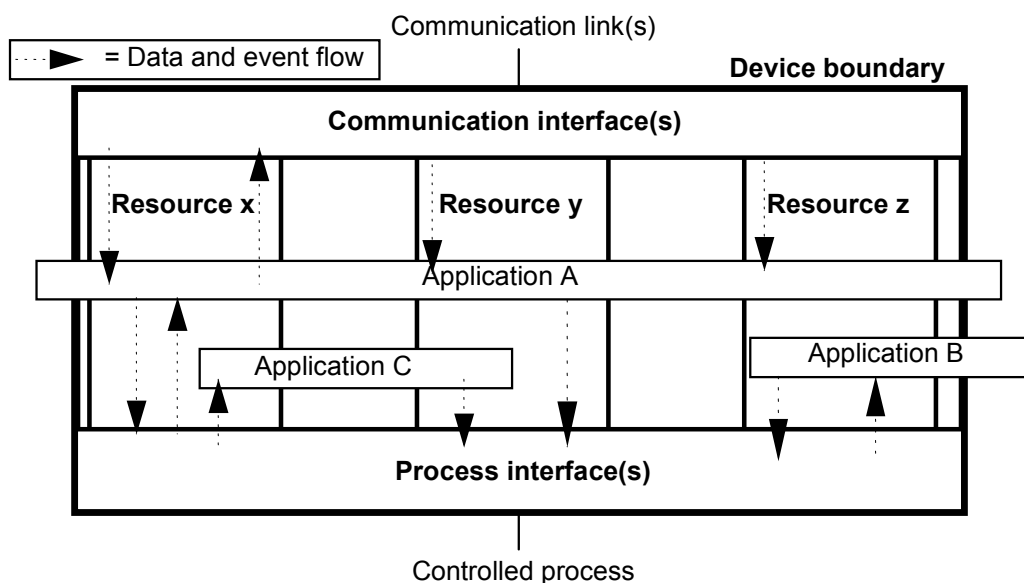
NOTE 2 A device that contains no resources is considered to be functionally equivalent to a *resource* as defined in 4.3.

A "process interface" provides a *mapping* between the physical process (analog measurements, discrete I/O, etc.) and the resources. Information exchanged with the physical process is presented to the resource as *data* or *events*, or both.

Communication *interfaces* provide a mapping between resources and the information exchanged via a communication *network*. Services provided by communication interfaces may include:

- presentation of communicated information to the resource as *data* or *events*, or both;
- additional services to support programming, *configuration*, diagnostics, etc.

Communication *links* may either be associated directly with a *device*, or with an instance of a specific *resource* type (communication resource), onto which part of the distributed application may or may not be mapped, depending on the resource type.



NOTE This figure shows a possible internal structure of Device 2 from Figure 1.

Figure 2 – Device model

4.3 Resource model

For the purposes of IEC 61499, a *resource* is considered to be a *functional unit*, which has independent control of its operation, contained in a *device*. It may be created, configured, parameterized, started up, deleted, etc., without affecting other resources.

NOTE 1 A resource is considered to be an *instance* of a corresponding resource *type*, defined as specified in 7.2.1.

NOTE 2 Although a resource has independent control of its operation, its operational states might need to be coordinated with those of other resources for the purposes of installation, test, etc.

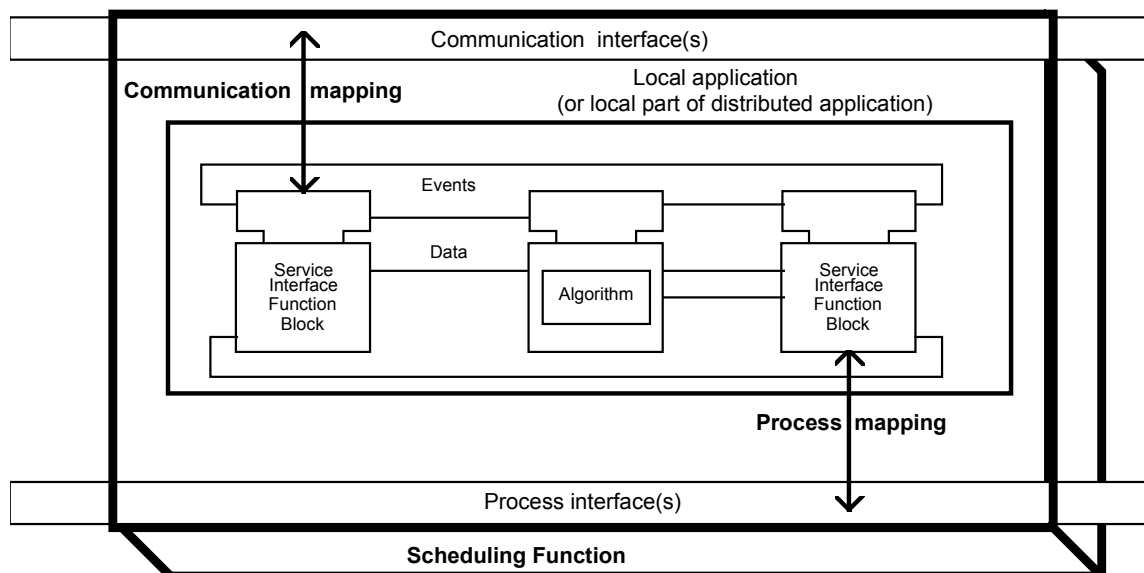
The *functions* of a resource are to accept *data* and/or *events* from the process and/or communication *interfaces*, process the data and/or events, and to return data and/or events to the process and/or communication interfaces, as specified by the *applications* utilizing the resource.

NOTE 3 Besides supporting the functions enumerated above, specific types of resources might represent the capability to implement interface functions such as process interfaces or lower layer communication services over communication links. Depending on the type of those resources, these services might or might not be the only ones they are able to provide.

NOTE 4 The consideration of other possible aspects of resources is beyond the scope of this standard.

As illustrated in Figure 3, a resource is modeled by the following.

- One or more "local applications" (or local parts of distributed applications). The *variables* and *events* handled in this part are *input* and *output variables* and events at *event inputs* and *event outputs* of *function blocks* that perform the *operations* needed by the application.
- A "process mapping" part whose function is to perform a *mapping* of *data* and *events* between *applications* and process interface(s). As shown in Figure 3, this mapping may be modeled by *service interface function blocks* specialized for this purpose.
- A "communication mapping" part whose function is to perform a *mapping* of *data* and *events* between *applications* and *communication interfaces*. As shown in Figure 3, this mapping may be modeled by *service interface function blocks* specialized for this purpose.
- A scheduling *function* which effects the execution of, and data transfer between, the function blocks in the applications, according to the timing and sequence requirements determined by:
 - a) the occurrence of events;
 - b) function block interconnections; and
 - c) scheduling information such as periods and priorities.



NOTE 1 This figure is illustrative only. Neither the graphical representation nor the location of function blocks is normative.

NOTE 2 Communication and process interfaces can be shared among resources.

Figure 3 – Resource model

4.4 Application model

For the purposes of this document, an *application* consists of a *function block network*, whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*.

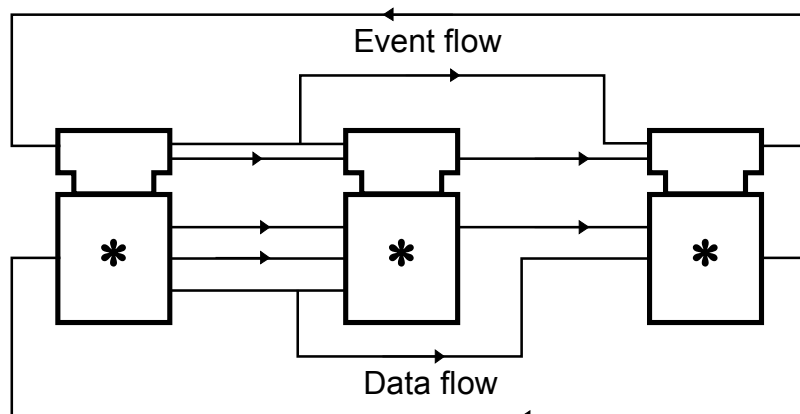
Subapplications are *instances* of *subapplication types*, which like applications consist of *function block networks*. Application names, subapplication and function block *instance names* may therefore be used to create a hierarchy of *identifiers* that can uniquely identify every *function block instance* in a *system*.

An application can be distributed among several *resources* in the same or different *devices*. A *resource* uses the causal relationships specified by the application to determine the appropriate responses to *events* which may arise from communication and process interfaces or from other functions of the resource. These responses may include:

- scheduling and *execution* of *algorithms*;
- modification of *variables*;
- generation of additional events;
- interactions with communication and process interfaces.

In the context of this document, applications are defined by *function block networks* specifying event and data flow among *function block* or *subapplication instances*, as illustrated in Figure 4. The event flow determines the scheduling and *execution* by the associated resource of the *operations* specified by each function block's *algorithm(s)*, according to the rules given in 5.2.2.

Standards, components and systems complying with this standard may utilize alternative means for scheduling of execution. Such alternative means shall be exactly specified using the elements defined in this standard.



NOTE 1 "*" represents function block or subapplication instances.

NOTE 2 This figure is illustrative only. The graphical representation is not normative.

Figure 4 – Application model

4.5 Function block model

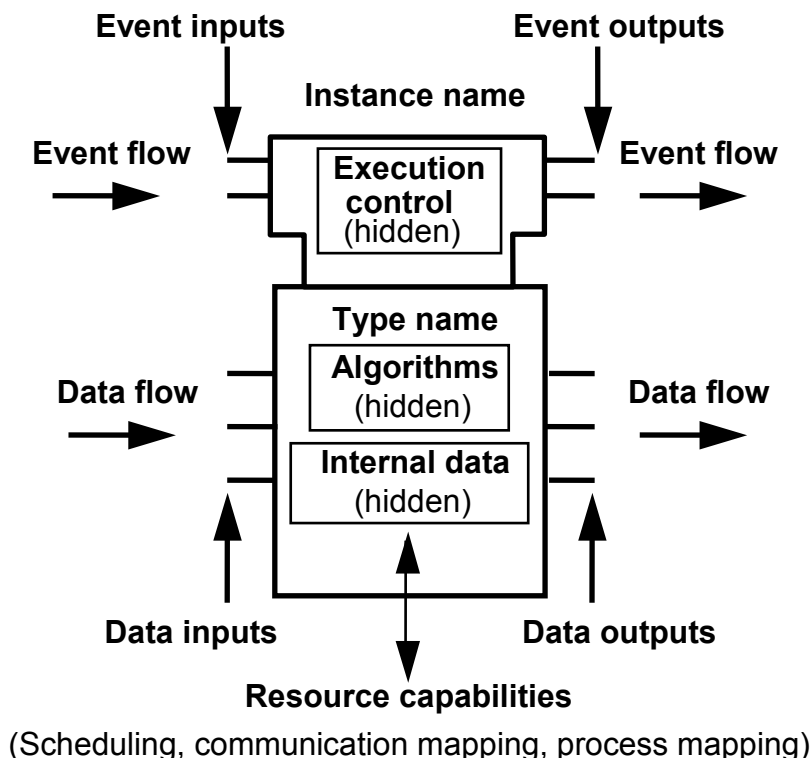
4.5.1 Characteristics of function block instances

A *function block* (*function block instance*) is a *functional unit* of *software* comprising an individual, named copy of the data structure specified by a *function block type*, which persists from one *invocation* of the function block to the next. The characteristics of function block instances are described in 4.5.1, and function block type specifications are described in 4.5.2.

A *function block instance* exhibits the following characteristic features as illustrated in Figure 5:

- its *type name* and *instance name*;
- a set of *event inputs*, each of which can receive *events* from an *event connection* which may affect the execution of one or more *algorithms*;
- a set of *event outputs*, each of which can issue *events* to an *event connection* depending on the execution of *algorithms* or on some other functional capability of the *resource* in which the function block is located;
- a set of *data inputs*, which may be *mapped* to corresponding *input variables*;
- a set of *data outputs*, which may be *mapped* to corresponding *output variables*;
- internal *data*, which may be *mapped* to a set of *internal variables*;
- functional characteristics which are determined by combining internal data or state information, or both, with a set of *algorithms*, functional capabilities of the associated *resource*, or both. These functional characteristics are defined in the function block's *type* specification.

NOTE Internal state information can be represented by *internal variables* or by an internal representation of an execution control state machine.



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

Figure 5 – Characteristics of function blocks

The algorithms contained within a function block are in principle invisible from the outside of the function block, except as described formally or informally by the provider of the function block. Additionally, the function block may contain internal *variables* or state information, or both, which persist between invocations of the function block's algorithms, but which are not accessible by data flow connections from the outside of the function block.

Access to internal variables and state information of function block instances may be provided by additional functional capabilities of the associated resource.

Means for specifying the causal relationships among event inputs, event outputs, and execution of algorithms are defined in Clauses 5 and 6.

4.5.2 Function block type specifications

A *function block type* is a *software* element which specifies the characteristics of all *instances* of the type, including:

- its *type name*;
- the number, names, type names and order of *event inputs* and *event outputs*.
- the number, names, *data type* and order of input, output and internal *variables*;

Mechanisms for the *declaration* of these characteristics are defined in 5.2.1.

In addition, the function block type specification defines the functionality of *instances* of the type. This functionality may be expressed as follows:

- For *basic function block types*, declaration mechanisms are provided in 5.2.1.3 for the specification of *algorithms*, which operate on the values of *input variables*, *output variables*, and *internal variables* to produce new values of *output variables* and *internal variables*. The associations among the *invocation* of algorithms and the occurrence of *events* at event inputs and outputs are expressed in terms of an *execution control chart* (ECC), using the declaration mechanisms defined in 5.2.1.4.
- The functionality of an *instance* of a *composite function block type* or a *subapplication type* is declared, using the mechanisms defined in 5.3.1 and 5.4.1 respectively, in terms of *data connections* and *event connections* among its *component function blocks* or subapplications and the event and data inputs and outputs of the composite function block or the subapplication.
- The functionality of an instance of a *service interface function block type* is described by a *mapping* of *service primitives* to *event inputs*, *event outputs*, *data inputs* and *data outputs*, using the declaration mechanisms defined in 6.1.
- Other means such as natural language text may be used for describing the functionality of a function block type; however, the specification of such means is beyond the scope of this standard.

4.5.3 Execution model for basic function blocks

As shown in Figure 6, the *execution* of *algorithms* for *basic function blocks* is invoked by the **execution control** portion of a *function block instance* in response to events at event inputs. This *invocation* takes the form of a request to the **scheduling function** of the associated *resource* to schedule the execution of the algorithm's *operations*. Upon completion of execution of an algorithm, the execution control generates zero or more events at *event outputs* as appropriate.

Events at event inputs are provided by connection to *event outputs* of other function block instances or the same function block instance. Events at these event outputs may be generated by execution control as described above, or by the "communication mapping", "process mapping", "scheduling", or other functional capability of the *resource*.

NOTE 1 Execution control in composite function blocks is achieved via event flow within the function block body.

Figure 6 depicts the order of events and algorithm execution for the case in which a single event input, a single algorithm, and a single event output are associated. The relevant times in this diagram are defined as follows:

- t_1 : relevant input variable values (i.e., those associated with the event input by the WITH qualifier defined in 5.2.1.2) are made available;
- t_2 : the event at the event input occurs;
- t_3 : the execution control function notifies the resource scheduling function to schedule an algorithm for execution;

- t_4 : algorithm execution begins;
- t_5 : the algorithm completes the establishment of values for the output variables associated with the event output by the WITH qualifier defined in 5.2.1.2;
- t_6 : the resource scheduling function is notified that algorithm execution has ended;
- t_7 : the scheduling function invokes the execution control function;
- t_8 : the execution control function signals an event at the event output.

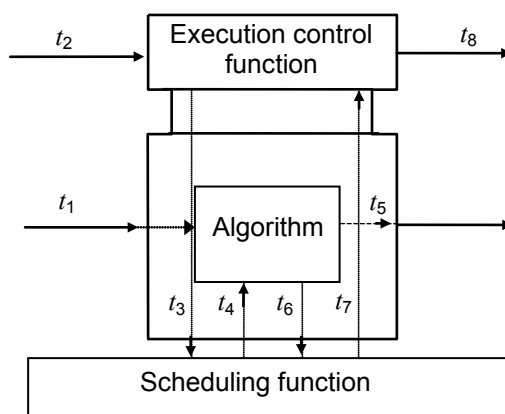
As shown in Figure 7, the significant timing delays in this case which are of interest in application design are:

$$T_{\text{setup}} = t_2 - t_1$$

$$T_{\text{start}} = t_4 - t_2 \text{ (time from event at event input to beginning of algorithm execution)}$$

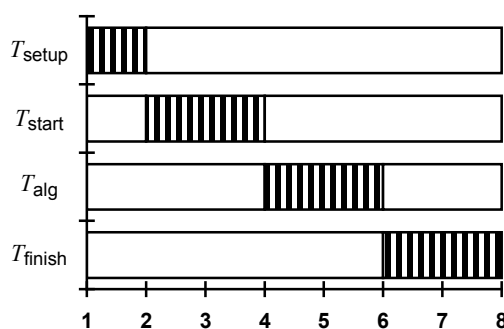
$$T_{\text{alg}} = t_6 - t_4 \text{ (algorithm execution time)}$$

$$T_{\text{finish}} = t_8 - t_6 \text{ (time from end of algorithm execution to event at event output)}$$



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

Figure 6 – Execution model



NOTE The axis labels 1,2,... in the above figure correspond to the times $t_1, t_2, \dots$ in Figure 6.

Figure 7 – Execution timing

Specific requirements for the graphical representation of *function block types* are given in 5.2.1.1.

NOTE 2 Depending on the problem to be solved, various requirements might exist for the synchronization of the values of *input variables* with the *execution of algorithms in order to ensure predictability of the results of algorithm execution*. Such requirements could include, for example:

- assurance that the values of variables used by an algorithm remain stable during the execution of the algorithm;
- assurance that the values of variables used by an algorithm correspond to the data present upon the occurrence of the event at the event input which caused the scheduling of the algorithm for execution;
- assurance that the values of variables used by all algorithms scheduled for execution in a function block correspond to the data present upon the occurrence of the event at the event input which caused the scheduling of the first such algorithm for execution.

NOTE 3 *Resources* might need to schedule the *execution of algorithms* in a *multitasking* manner. The specification of attributes to facilitate such scheduling is described in Annex G.

4.6 Distribution model

As illustrated in Figure 8a, an *application* or *subapplication* can be distributed by allocating its *function block instances* to different *resources* in one or more *devices*. Since the internal details of a function block are hidden from any application or subapplication utilizing it, a function block shall form an atomic unit of distribution. That is, all the elements contained in a given function block instance shall be contained within the same resource.

The functional relationships among the function blocks of an application or subapplication shall not be affected by its distribution. However, in contrast to an application or subapplication confined to a single resource, the timing and reliability of communications functions will affect the timing and reliability of a distributed application or subapplication.

The following clauses apply when applications or subapplications are distributed among multiple resources:

- Clause 6 defines the requirements for communication services to support distribution of applications or subapplications among multiple devices;
- Clause 7 defines the requirements for the case where multiple applications or subapplications are distributed among multiple resources and devices.

4.7 Management model

Figures 8b and 8c provide a schematic representation of the management of *resources* and *devices*. Figure 8b illustrates a case in which a *management resource* provides shared facilities for management of other *resources* within a device, while Figure 8c illustrates the distribution of management services among resources within a device. Management *applications* may be modeled using **implementation-dependent service interface function blocks** and *communication function blocks*.

NOTE 1 6.3 defines *service interface function block types* for management of *applications*, and IEC 61499-2 provides examples of their usage.

NOTE 2 *Management applications* might contain *service interface function block instances* representing *device* or *resource instances* for the purpose of querying or modifying device or resource *parameters*.

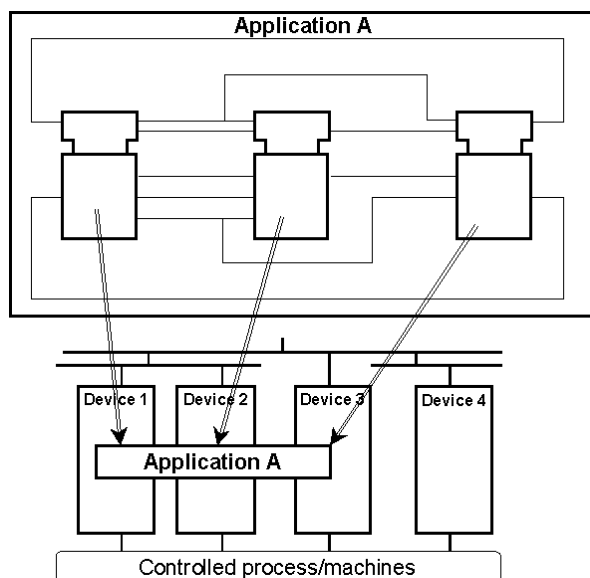


Figure 8a – Distribution model

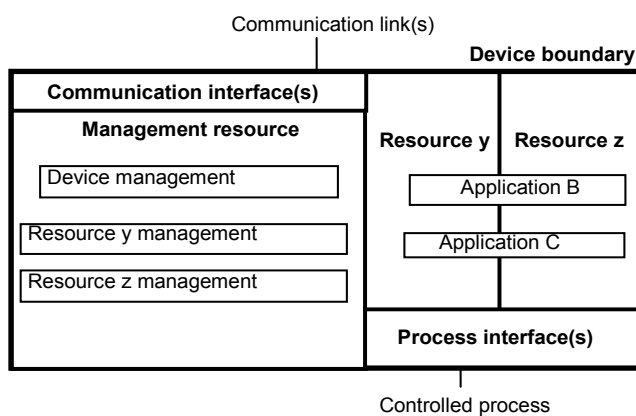


Figure 8b – Shared management model

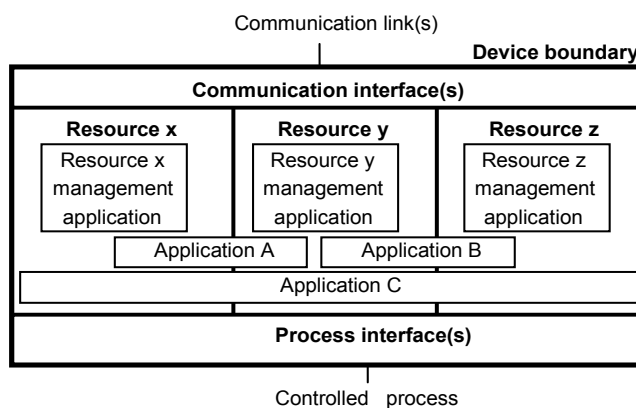


Figure 8c – Distributed management model

Figure 8 – Distribution and management models

4.8 Operational state models

Any given *system* has to be designed, commissioned, operated and maintained. This is modeled through the concept of the system "life cycle". In turn, a system is composed of several *functional units* such as *devices*, *resources*, and *applications*, each of which has its own life cycle.

Different actions may have to be performed to support *functional units* at each step of the life cycle. To characterize which action can be done and maintain integrity of functional units, "operational states" should be defined, e.g., OPERATIONAL, CONFIGURABLE, LOADED, STOPPED, etc.

Each operational state of a functional unit specifies which actions are authorized, together with an expected behavior.

A system may be organized in such a way that certain functional units may possess or acquire the right of modifying the operational states of other functional units.

Examples of the use of operational states are:

- a functional unit in a RUNNING state, i.e., in execution, may not be able to receive a download action;
- a distributed functional unit may need to maintain a consistent operational state across its components and develop a strategy to propagate changes of operational state through them.

Specific operational states for managed *function block instances* are defined in 6.3.2.

5 Specification of function block, subapplication and adapter interface types

5.1 Overview

As illustrated in Figure 9, Clause 5 defines the means for the type specification of three kinds of blocks:

- Subclause 5.2 defines the means for specifying and determining the behavior of instances of *basic function block types*, as illustrated in Figure 9a. In this type of function block, execution control is specified by an *execution control chart (ECC)*, and the *algorithms* to be executed are declared as specified in compliant Standards as defined in IEC 61499-4.
- Subclause 5.3 defines the means for specifying *composite function block types*, as illustrated in Figure 9b. In this type of function block, algorithms and their execution control are specified through event and data connections in one or more *function block networks*.
- Subclause 5.4 defines the means for specifying *subapplication types*, as illustrated in Figure 9c. In this type of block, algorithms and their execution control are specified as for composite function block types, but with the specific property that *component function blocks* of subapplications may be distributed among several *resources*. Subapplications may be nested, such that the body of a subapplication may also contain *component subapplications*.

Other means may be used for describing the behavior of instances of a function block type. The specification of such means is beyond the scope of this standard; therefore it is required that when such means are used, an unambiguous *mapping* shall be given between their terms and concepts and the corresponding terms and concepts of this standard.

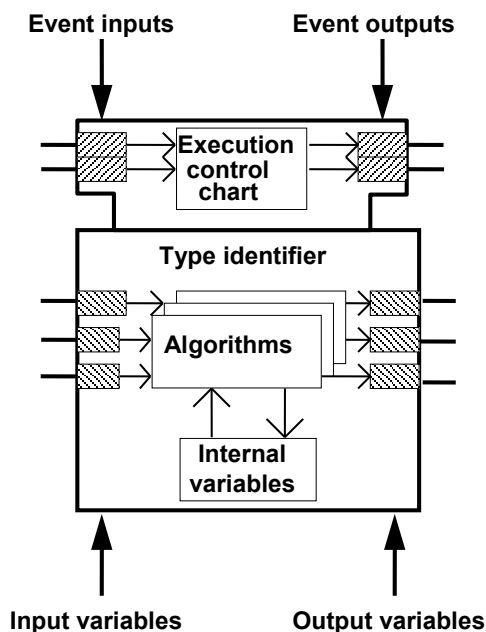


Figure 9a – Basic function block (5.2)

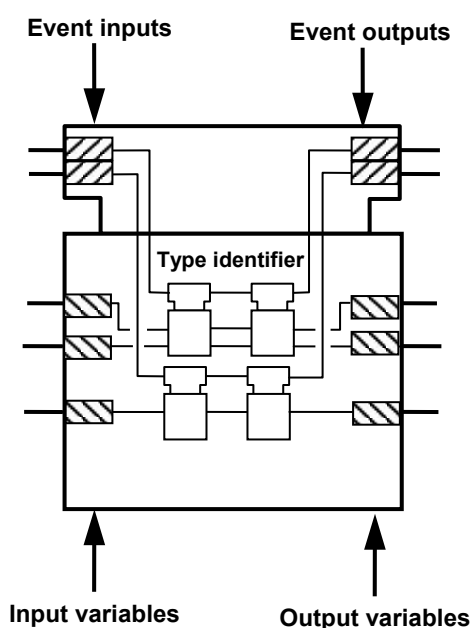
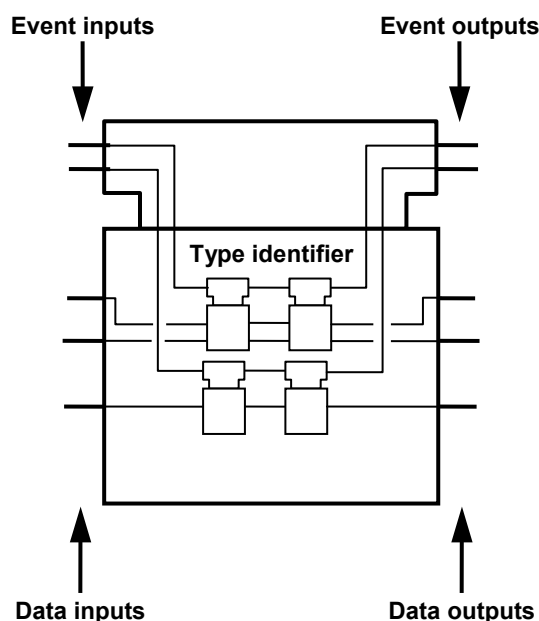


Figure 9b – Composite function block (5.3)



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

Figure 9c – Subapplications (5.4)

Figure 9 – Function block and subapplication types

5.2 Basic function blocks

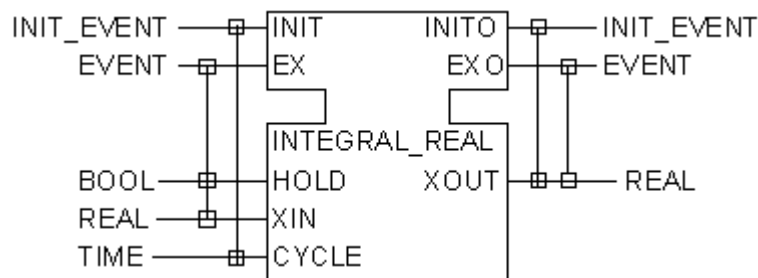
5.2.1 Type declaration

5.2.1.1 General

A *basic function block* utilizes an *execution control chart (ECC)* to control the *execution* of its *algorithms*.

As illustrated in Figure 10, a *basic function block type* can be declared textually according to the syntax specified in Clause B.2 or graphically according to the following rules:

- the function block *type name* is shown at the top center of the lower portion of the block;
- the names and *type declarations* of *input variables* and *socket adapters* are shown at the left edge of the lower portion of the block;
- the names and *type declarations* of *output variables* and *plug adapters* are shown at the right edge of the lower portion of the block;
- the *interface* of the function block type to *events* is declared in the upper portion of the block as specified in 5.2.1.2;
- the *algorithms* associated with the function block type are declared as specified in 5.2.1.3;
- control of the *execution* of the associated algorithms is declared as specified in 5.2.1.4.



NOTE 1 See Annex F for a textual declaration of this example.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 10 – Basic function block type declaration

5.2.1.2 Event interface declaration

As shown in Figure 10, the *interface* of a *basic function block type* to *events* can be declared textually according to the syntax given in Clause B.2, or graphically according to the following rules.

- Event interfaces* are located in a distinct area at the top of the block.
- Event input* names are shown at the left-hand side of the upper portion of the block.
- Event output* names are shown at the right-hand side of the upper portion of the block.
- Event types* are shown outside the block adjacent to their associated event inputs or outputs.

NOTE 1 If no event type is given for an event input or output, it is considered to be of the default type EVENT.

NOTE 2 An event output of type EVENT can be connected to an event input of any type, and an event input of type EVENT can receive an event of any type.

NOTE 3 An event output of any type other than EVENT can only be connected to an event input of the same type or of type EVENT.

NOTE 4 An event type is implicitly declared by its use in an event declaration.

As illustrated in Figure 10 and Annex F, the **WITH** qualifier or a graphical equivalent shall be used to specify an association among *input variables* or *output variables* and an *event* at the associated *event input* or *event output*, respectively.

Each *input variable* and *output variable* appears in zero or more **WITH** clauses or their graphical equivalents.

NOTE 5 This information can be used to determine the required communication *services* when *configuring* a distributed *application* as described in Clause 7.

NOTE 6 An input variable that does not appear in any **WITH** clause cannot be connected with an output variable of another function block. The values of such variables either remain at their declared initial values or are established by management commands such as WRITE, as described in 6.3.2.

NOTE 7 An output variable that does not appear in any **WITH** clause can be connected to an input variable of another function block or can be "read" by management commands such as **READ**, as described in 6.3.2.

NOTE 8 See 4.5.3 for an application of the **WITH** qualifier to the execution model of a basic function block.

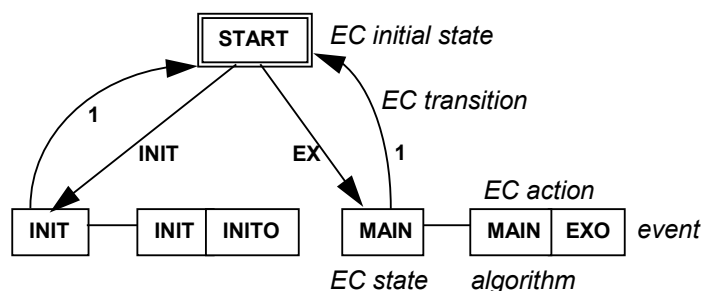


Figure 11 – ECC example

5.2.2 Behavior of instances

5.2.2.1 Initialization

Initialization of a basic function block *instance* by a *resource* shall be functionally equivalent to the following procedure:

- The value of each *input*, *output*, and *internal variable* shall be initialized to the corresponding initial value given in the function block *type* specification. If no such initial value is defined, the value of the variable shall be initialized to the default initial value defined for the data type of the variable.
- Any additional algorithm-specific initializations shall be performed; for example, all *initial steps* of IEC 61131-3 *Sequential Function Charts (SFCs)* shall be activated and all other *steps* shall be deactivated.
- The *EC initial state* of the function block's *Execution Control Chart (ECC)* shall be activated, all other *EC states* shall be deactivated, and the ECC operation state machine defined in 5.2.2.2 shall be placed in its initial (*s0*) state.

NOTE The conditions under which a resource performs such initialization are **implementation-dependent**.

The function block *type* may also specify an initialization *algorithm* to be performed upon the occurrence of an appropriate event, for example the **INIT** algorithm shown in Figure 11. An *application* can then specify the conditions under which this algorithm is to be executed, for example by connecting an output of an instance of the **E_RESTART** type defined in Annex A to an appropriate event input, for example the **INIT** input shown in Figure 10.

5.2.2.2 Algorithm invocation

Execution of an *algorithm* associated with a *function block instance* is *invoked* by a request to the **scheduling function** of the *resource* to schedule the execution of the algorithm's *operations*.

NOTE 1 The operations performed by an algorithm can vary from one execution to the next due to changed internal states of the function block, even though the function block may have only a single algorithm and a single event input triggering its execution.

Algorithm invocation for an instance of a *basic function block type* shall be accomplished by the functional equivalent of the operation of its *execution control chart (ECC)*. The operation of the ECC shall exhibit the behavior defined by the state machine in Figure 12 and Table 1.

NOTE 2 It is a consequence of this model that an occurrence of an event at an event input will not cause a transition containing the event to be crossed, if the transition is not associated with the currently active state, i.e., if the event is not relevant in the given state. However, *sampling* of the input variables associated to the event by a **WITH** construct will occur in any case.

5.2.2.3 Algorithm declaration

As shown in Annex F, *algorithms* associated with a *basic function block type* may be included in the function block type declaration according to the rules for declaration of the function block type specification given in Annex B. Other means may also be used for the specification of the identifiers and bodies of algorithms; however, the specification of such means is beyond the scope of this standard.

The declaration of an algorithm may include the declaration of temporary variables that:

- are only visible in the body of the algorithm;
- are initialized upon each invocation of the algorithm;
- may be used and modified during execution of the algorithm; and
- do not have values that persist between executions of the algorithm.

5.2.2.4 Declaration of algorithm execution control

The sequencing of algorithm invocations for *basic function block types* may be declared in the function block type specification. If the algorithms of a basic function block type are defined as specified in 5.2.1.3 (or otherwise identified), then the sequencing of algorithm invocation for such a function block can be in the form of an *Execution Control Chart (ECC)* consisting of *EC states*, *EC transitions*, and *EC actions*. These elements are represented and interpreted as follows:

- a) the ECC is included in an *execution control* section of the function block type declaration, considered to reside in the upper portion of the block;
- b) the ECC shall contain exactly one *EC initial state*, represented graphically as a double-outlined shape with an associated *identifier*. The EC initial state shall have no associated EC actions;
- c) the ECC shall contain one or more *EC states*, represented graphically as single-outlined shapes, each with an associated *identifier*;
- d) the ECC can utilize but not modify variables declared in the function block type specification;
- e) an *EC state* can have zero or more associated *EC actions*. The association of the EC actions with the EC state can be expressed in graphical or textual form;
- f) the *algorithm* (if any) associated with an EC action, and the *event* (if any) to be issued on completion of the algorithm, shall be expressed in graphical or textual form;
- g) an *EC transition* is represented graphically or textually as a directed link from one EC state to another (or to the same state);
- h) each EC transition shall have an associated transition condition, containing a reference to an *event*, a *guard condition*, or both, expressed in the syntax defined for the non-terminal `ec_transition_condition` in B.2.1.

Figure 11 illustrates the elements of an ECC. Similar textual declarations using the syntax of Clause B.2 are given in Annex F.

NOTE 1 The notation 1 (one), illustrated in Figure 11, is considered to be equivalent to [TRUE] representing a transition condition with no associated event and a guard condition that is always TRUE.

NOTE 2 In this restricted domain, the same symbol (e.g., INIT) can be used to represent an EC state and algorithm name, since the referent of the symbol can be inferred easily from its usage.

NOTE 3 The text in *italics* is not part of the ECC.

NOTE 4 One-to-one association of events with algorithms, as illustrated in this figure, is frequently encountered but is not the only possible usage. See Table A.1 for examples of other usages: The E_SPLIT block shows an association of two event outputs with one state but no algorithms; E_MERGE shows an association of one output event but no algorithms with two event inputs; E_DEMUX shows any of several algorithms associated with a single input event; etc.

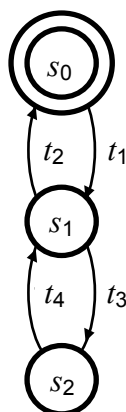


Figure 12 – ECC operation state machine

Table 1 – States and transitions of ECC operation state machine

State		Operations
s_0		--
s_1		evaluate transitions ^{c,e}
s_2		perform actions ^{d,e}
Transition	Condition	Operations
t_1	an input event occurs ^a	Sample inputs ^{b,e}
t_2	no transition is crossed	
t_3	a transition is crossed	
t_4	actions completed	

^a The resource shall ensure that no more than one input event occurs at any given instant in time.

^b This operation consists of *sampling* (or its functional equivalent) of the input variables associated with the current input event by a WITH declaration as described in 5.2.1.2.

^c This operation consists of evaluating the transition conditions at the EC transitions following the active EC state and crossing the first EC transition (if any) for which a TRUE guard_condition as defined in B.2.1 is found, according to the following rules:

- 1 "Crossing the EC transition" shall consist of deactivating its predecessor EC state and activating its successor EC state.
- 2 The order in which the transition conditions are evaluated shall correspond to the order in which the transitions are declared as defined in B.2.1, or equivalently in the XML syntax defined in IEC 61499-2.
- 3 The guard_condition of a transition condition containing only an event_input_name shall have the default value TRUE.
- 4 If state s_1 was entered via t_1 , only transition conditions associated with the current input event via its event_input_name as defined in B.2.1, or transition conditions with no event associations, shall be evaluated.
- 5 If state s_1 was entered via t_4 , only transition conditions with no event associations shall be evaluated.

^d This operation consists of, for each EC action associated with the active EC step, executing the associated algorithm, if any, and issuing an event at the associated event output, if any. The order in which the actions are performed corresponds to the order in which they appear graphically from top to bottom, or to the order in which they are declared following the textual syntax defined in B.2.1, or equivalently in the XML syntax defined in IEC 61499-2.

^e All operations performed from an occurrence of transition t_1 to an occurrence of t_2 shall be implemented as a *critical region* with a lock on the function block instance.

5.2.2.5 Algorithm execution

Algorithm *execution* in a basic function block shall consist of the execution of a finite sequence of *operations* determined by **implementation-dependent** rules appropriate to the

language in which the algorithm is written, the *resource* in which it executes, and the domain to which it applies. Algorithm execution terminates after execution of the last operation in this sequence.

If an algorithm implements a state machine, repeated executions of the algorithm are necessary to recognize or perform state changes. Normally there is no association between those state changes and the completion of the algorithm. Such associations have to be created by the event output generation facilities described in 5.2.2.2.

5.3 Composite function blocks

5.3.1 Type specification

The declaration of *composite function block types* shall follow the rules given in 5.2.1 with the exception that *event inputs* and *event outputs* of the *component function blocks* can be interconnected with the event inputs and event outputs of the composite function block to represent the sequencing and causality of function block invocations. The following rules shall apply to this usage:

- a) Each event input of the composite function block is connected to exactly one event input of exactly one component function block, or to exactly one event output of the composite function block, with the exception that the graphical shorthand for event splitting shown in Figure A.1 may be employed.
- b) Each event input of a component function block is connected to no more than one event output of exactly one other component function block, or to no more than one event input of the composite function block, with the exception that the graphical shorthand for event merging shown in Figure A.1 may be employed.
- c) Each event output of a component function block is connected to no more than one event input of exactly one other component function block, or to no more than one event output of the composite function block, with the exception that the graphical shorthand for event splitting shown in Figure A.1 may be employed.
- d) Each event output of the composite function block is connected from exactly one event output of exactly one component function block, or from exactly one event input of the composite function block, with the exception that the graphical shorthand for event merging shown in Figure A.1 may be employed.
- e) Use of the *WITH* qualifier in the declaration of event inputs of composite function block types is required. Use of the *WITH* qualifier may result in the *sampling* of the associated data inputs as in the case of basic or service interface function blocks, or software tools may provide means of elimination of redundant sampling in the implementation phase.
- f) *Instances of subapplication types* as defined in 5.4 shall not be used in the specification of a composite function block type.

Data inputs and *data outputs* of the *component function blocks* can be interconnected with the data inputs and data outputs of the composite function block to represent the flow of data within the composite function block. The following rules shall apply to this usage:

- Each data input of the composite function block can be connected to zero or more data inputs of zero or more component function blocks, or to zero or more data outputs of the composite function block, or both.
- Each data input of a component function block can be connected to no more than one data output of exactly one other component function block, or to no more than one data input of the composite function block.
- Each data output of a component function block can be connected to zero or more data inputs of zero or more component function blocks, or to zero or more data outputs of the composite function block, or both.
- Each data output of the composite function block shall be connected from exactly one data output of exactly one component function block, or from exactly one data input of the composite function block.

NOTE 1 If an element declared in a `VAR_INPUT...END_VAR` or `VAR_OUTPUT...END_VAR` construct is associated with an input or output event, respectively, by a `WITH` construct, this will result in the creation of an associated input or output variable, respectively, as in the case of basic function block types. If such an element is not associated with an input or output event, then the associated data flow is passed directly to or from the component function blocks via the connections described above.

NOTE 2 The rules for interconnection of the event and variable inputs and outputs of *plugs* and *sockets* in the body of the composite function block are the same as for the interconnection of the inputs and outputs of the *component function blocks*. See 5.5 for further requirements regarding *adapter interfaces*.

Figure 13 illustrates the application of these rules to the example `PI_REAL` function block. Figure 13a shows the graphical representation of the external interfaces and 13b shows the graphical construction of its body. Figure 14 shows the interfaces and execution control for the function block type `PID_CALC` used in the body of the `PI_REAL` example.

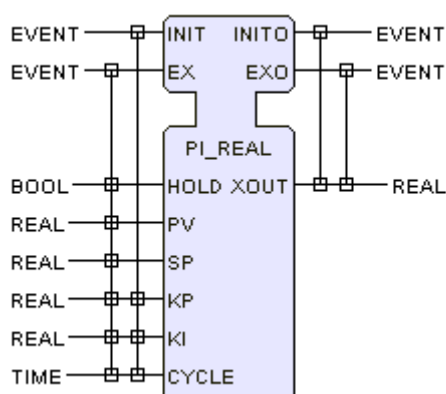


Figure 13a – External interface

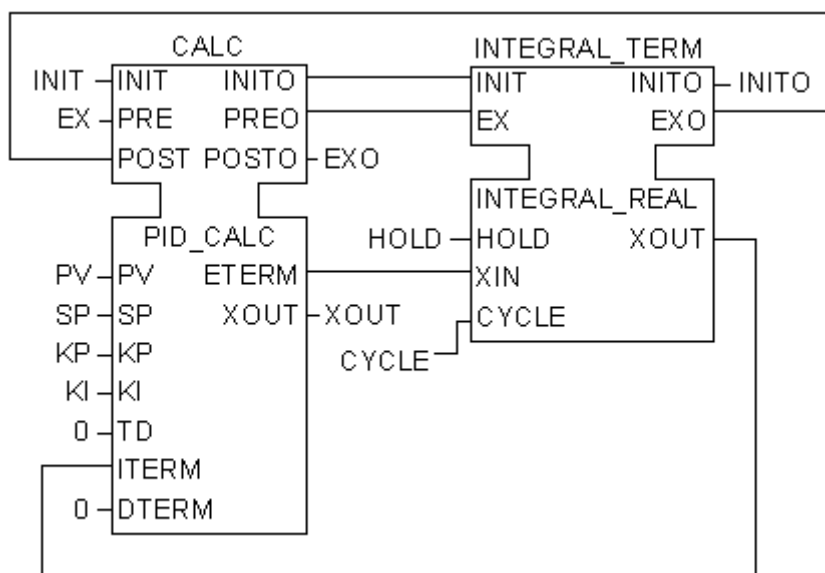


Figure 13b – Graphical body

NOTE 1 A full textual declaration of this function block type is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 13 – Composite function block `PI_REAL` example

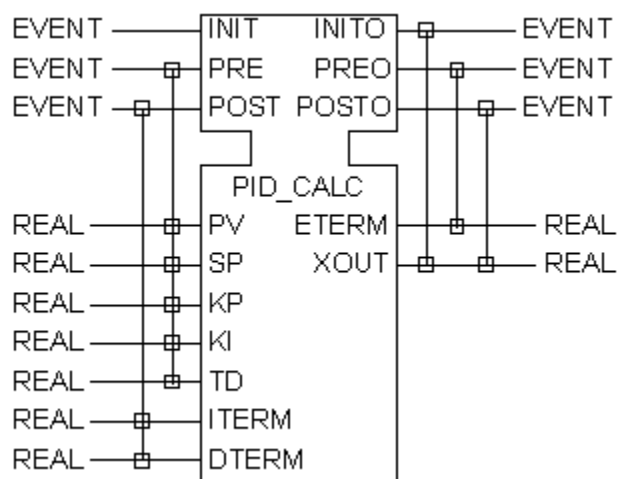


Figure 14a – External interface

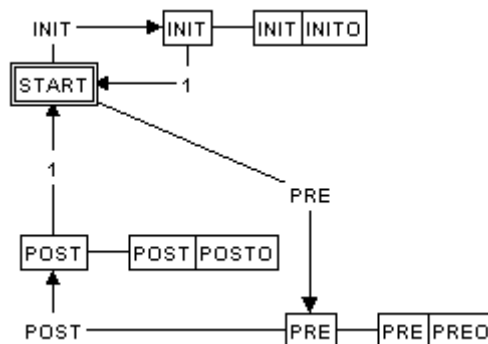


Figure 14b – Execution control

NOTE This example is illustrative only. Details of the specification are not normative.

Figure 14 – Basic function block `PID_CALC` example

5.3.2 Behavior of instances

Invocation and execution of component function blocks in composite function blocks shall be accomplished as follows.

- If an *event input* of the composite function block is connected to an *event output* of the block, occurrence of an *event* at the event input shall cause the generation of an event at the associated event output.
- If an event input of the composite function block is connected to an event input of a component function block, occurrence of an event at the event input of the composite function block shall cause the scheduling of an invocation of the execution control function of the component function block, with an occurrence of an event at the associated event input of the component function block.
- If an event output of a component function block is connected to an event input of a second component function block, occurrence of an event at the event output of the first block shall cause the scheduling of an invocation of the execution control function of the second block, with an occurrence of an event at the associated event input of the second block.
- If an event output of a component function block is connected to an event output of the composite function block, occurrence of an event at the event output of the component block shall cause the generation of an event at the associated event output of the composite function block.

Initialization of instances of composite function blocks shall be equivalent to initialization of their component function blocks according to the provisions of 5.2.2.1.

5.4 Subapplications

5.4.1 Type specification

The declaration of *subapplication types* is similar to the declaration of *composite function block types* as defined in 5.3.1, with the exception that the delimiting keywords shall be `SUBAPPLICATION...END_SUBAPPLICATION`. The following rules shall apply to this usage:

- a) The `WITH` qualifier is not used in the declaration of event inputs and event outputs of *subapplication types*.
- b) Each event input of the subapplication shall be connected to exactly one event input of exactly one component function block or component subapplication, or to exactly one event output of the subapplication.
- c) Each event input of a component function block or component subapplication is connected to no more than one event output of exactly one other component function block or component subapplication, or to no more than one event input of the subapplication.
- d) Each event output of a component function block or component subapplication is connected to no more than one event input of exactly one other component function block or component subapplication, or to no more than one event output of the subapplication.
- e) Each event output of the subapplication is connected from exactly one event output of exactly one component function block or component subapplication, or from exactly one event input of the subapplication.

NOTE 1 Component function blocks can include instances of the event processing blocks defined in Annex A, for example to "split" events using instances of the `E_SPLIT` block, to "merge" events using instances of the `E_MERGE` block, or for both cases, using the equivalent graphical shorthand.

Data inputs and data outputs of the *component function blocks* or *component subapplications* can be interconnected with the data inputs and data outputs of the subapplication to represent the flow of data within the subapplication. The following rules shall apply to this usage:

- Each data input of the subapplication can be connected to zero or more data inputs of zero or more component function blocks or component subapplications, or to zero or more data outputs of the subapplication, or both.
- Each data input of a component function block or component subapplication can be connected to no more than one data output of exactly one other component function block or component subapplication, or to no more than one data input of the subapplication.
- Each data output of a component function block or component subapplication can be connected to zero or more data inputs of zero or more component function blocks or component subapplications, or to zero or more data outputs of the subapplication, or both.
- Each data output of the subapplication shall be connected from exactly one data output of exactly one component function block or component subapplication, or from exactly one data input of the subapplication.

NOTE 2 Although the `VAR_INPUT...END_VAR` and `VAR_OUTPUT...END_VAR` constructs are used for the declaration of the data inputs and outputs of subapplication types, this does not result in the creation of input and output variables; the data flow is instead passed to the component function blocks or component subapplications via the connections described above.

NOTE 3 The rules for interconnection of the event and variable inputs and outputs of *plugs* and *sockets* in the body of the subapplication are the same as for the interconnection of the inputs and outputs of the *component function blocks*. See 5.5 for further requirements regarding *adapter interfaces*.

EXAMPLE Figure 15 illustrates the application of these rules to the example `PI_REAL_APPL` subapplication. Figure 15a shows the graphical representation of its external interfaces and Figure 15b shows the graphical construction of its body. The body of the `PI_REAL_APPL` subapplication example uses the function block type `PID_CALC` from the composite function block example in 5.3.1, which is shown in Figure 14.

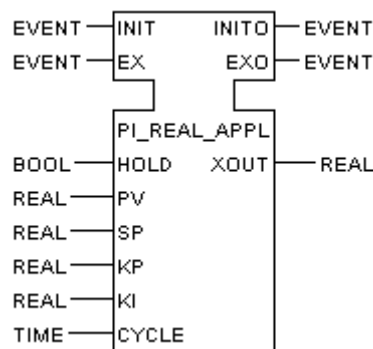


Figure 15a – External interface

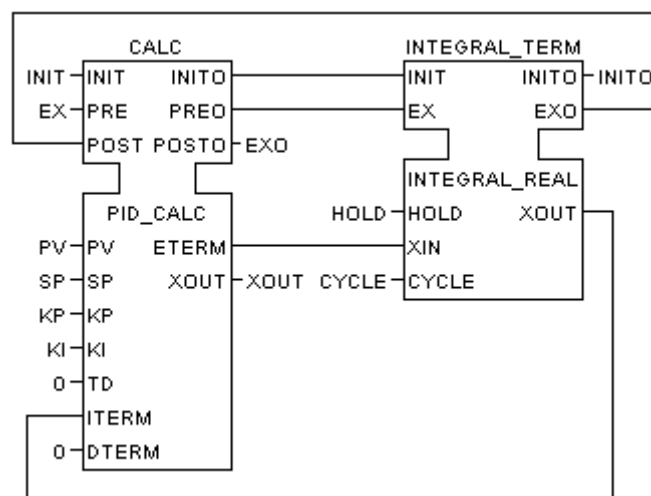


Figure 15b – Graphical body

NOTE 1 A full textual declaration of this subapplication type is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

Figure 15 – Subapplication PI_REAL_APPL example

5.4.2 Behavior of instances

Invocation of the operations of component function blocks or component subapplications within subapplications shall be accomplished as follows:

- If an *event input* of the subapplication is connected to an *event output* of the block, occurrence of an *event* at the event input shall cause the generation of an event at the associated event output.
- If an event input of the subapplication is connected to an event input of a component function block or component subapplication, occurrence of an event at the event input of the subapplication shall cause the scheduling of an invocation of the execution control function of the component function block or component subapplication, with an occurrence of an event at the associated event input of the component function block or component subapplication.
- If an event output of a component function block or component subapplication is connected to an event input of a second component function block or component subapplication, occurrence of an event at the event output of the first block shall cause the scheduling of an invocation of the execution control function of the second block, with an occurrence of an event at the associated event input of the second block.

- d) If an event output of a component function block or component subapplication is connected to an event output of the subapplication, occurrence of an event at the event output of the component block shall cause the generation of an event at the associated event output of the subapplication.

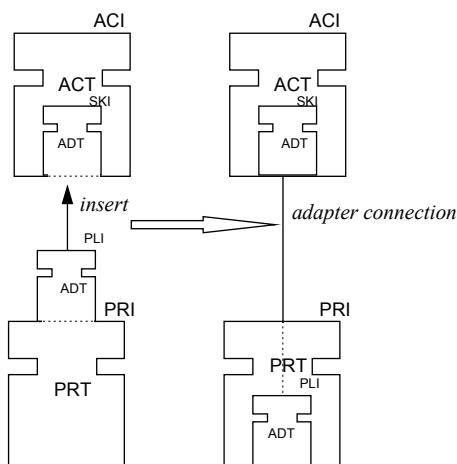
Since subapplications do not explicitly create variables, no specific initialization procedures are applicable to subapplication instances.

5.5 Adapter interfaces

5.5.1 General principles

Adapter interfaces can be used to provide a compact representation of a specified set of event and data flows. As illustrated in Figure 16, an *adapter interface type* provides a means for defining a subset (the *plug adapter*) of the *inputs* and *outputs* of a *provider* function block which can be inserted into a matching subset of corresponding *outputs* and *inputs* (the *socket adapter*) of an *acceptor* function block. Thus, the adapter interface represents the event and data paths by which the provider supplies a *service* to the acceptor, or vice versa, depending on the patterns of provider/acceptor interactions, which may be represented by sequences of *service primitives* as described in 6.1.3.

NOTE A given *function block type* might function as a *provider*, an *acceptor*, or both, or neither, and may contain more than one *plug* or *socket* instance of one or more *adapter interface types*.



Key

PRT	Provider type
PRI	Provider instance
ACT	Acceptor type
ACI	Acceptor instance
ADT	Adapter type
PLI	Plug instance
SKI	Socket instance

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

Figure 16 – Adapter interfaces – Conceptual model

5.5.2 Type specification

An *adapter interface type declaration* shall define only the *interface type* name and its contained *event* and *data interfaces*. These are defined graphically or textually in the same manner as the *type name*, *event interfaces* and *data interfaces* of a *basic function block type* as defined in 5.2.1.1 and 5.2.1.2, with the exception that the keywords for beginning and ending the textual type declaration shall be `ADAPTER...END_ADAPTER`. Textual syntax for the declaration of adapter interfaces is given in Clause B.7.

EXAMPLE The adapter interface illustrated in Figure 17 represents the operation of transferring a workpiece from an "upstream" piece of transfer equipment represented by a *provider* of the *plug* adapter to a "downstream" piece of equipment represented by an *acceptor* with a corresponding *socket* adapter. As illustrated in Figure 17b, the typical operation of this interaction consists of the following sequence:

- An event in the upstream equipment, e.g., arrival of a workpiece at the unload position, causes a LD event, typically interpreted as a "load" command, to be transmitted to the downstream equipment. Associated with this event is a sensor value WO, indicating whether a workpiece is actually present for transfer, plus some measured property or set of properties of the workpiece, in this case its color.
- A subsequent event in the downstream equipment, e.g., completion of the load setup, causes an UNLD event, typically interpreted as a command to release the workpiece, to be sent to the upstream equipment.
- Subsequently a CNF event, typically interpreted as confirmation of the workpiece release, is passed from the upstream to the downstream equipment to complete the operation. At this point the WO output is typically FALSE and the value of the WKPC output has no significance.

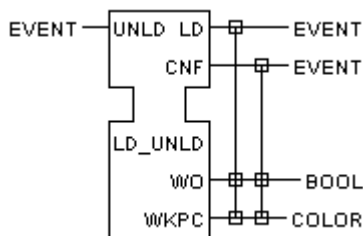


Figure 17a – Interface

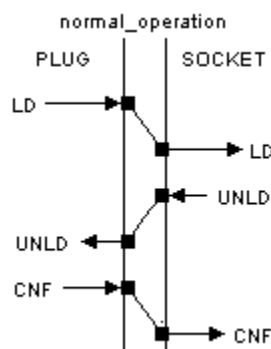


Figure 17b – Service sequence

NOTE 1 A full textual declaration of this adapter type is given in Annex F.

NOTE 2 This example is illustrative only. Details of the specification are not normative.

NOTE 3 See 6.1.2 for an explanation of service sequences.

Figure 17 – Adapter type declaration – graphical example

5.5.3 Usage

The usage of *adapter interface types* and *instances* shall be according to the following rules:

- Adapter interface instances to be used as *plugs* in instances of a *function block type* shall be declared in its *type declaration* in a `PLUGS...END_PLUGS` block, declaring the *instance name* and *adapter interface type* of each plug. In the graphical representation of *function block types* and *instances*, plugs are shown as *output variables* with specialized textual or graphical indication to show that they are not ordinary output variables.
- Adapter interface instances to be used as *sockets* in instances of a *function block type* shall be declared in its *type declaration* in a `SOCKETS...END_SOCKETS` block, declaring the *instance name* and *adapter interface type* of each socket. In the graphical representation of *function block types* and *instances*, sockets are shown as *input variables* with specialized textual or graphical indication to show that they are not ordinary input variables.
- Inputs* and *outputs* of a *plug* shall be used within its *function block type declaration* in the same manner as inputs and outputs of the function block.
- Inputs* and *outputs* of a *socket* shall be used within its *function block type declaration* in the same manner as *outputs* and *inputs* of the function block, respectively.
- Insertion of *plugs* into *sockets* shall be specified in an `ADAPTER_CONNECTIONS...END_CONNECTIONS` block in the *declaration* of the *application*, *subapplication*, *resource type*, *resource instance*, or *composite function block type* containing the respective *provider* and *acceptor* instances.

- f) In the body of a *composite function block type* or *subapplication*, a *socket* is represented as a *function block* with the same inputs and outputs as the corresponding *adapter interface type*. Similarly, in this case a *plug* is represented as a function block with the inputs and outputs of the corresponding adapter interface type reversed.
- g) Insertion of plugs into sockets shall be subject to the following constraints:
 - 1) a plug can only be inserted into a socket of the same *adapter interface type*;
 - 2) a plug can only be inserted into zero or one socket at a time;
 - 3) a socket can only accept zero or one plug at a time;
 - 4) a plug can only be inserted in a socket if both are in the same *composite function block, resource, application or subapplication*.

A connection from a plug to a socket may be shown in an *application* or *subapplication* even though the corresponding function block instances may be *mapped* to separate *resources*. In this case appropriate means, such as communication service interface function blocks as described in 6.2, shall be used to implement the corresponding transfer of events and data among resources.

Management function blocks as described in 6.3 may provide facilities for the dynamic creation, deletion, and querying of adapter connections.

EXAMPLE 1 An instance of the XBAR_MVCA type illustrated in Figure 18 acts as both a provider of a plug interface (LDU_PLG) and an acceptor with a socket interface (LDU_SKT). In so doing, it serves to abstract and encapsulate the interactions of an instance of the XBAR_MVC type with "upstream" and "downstream" functional units.

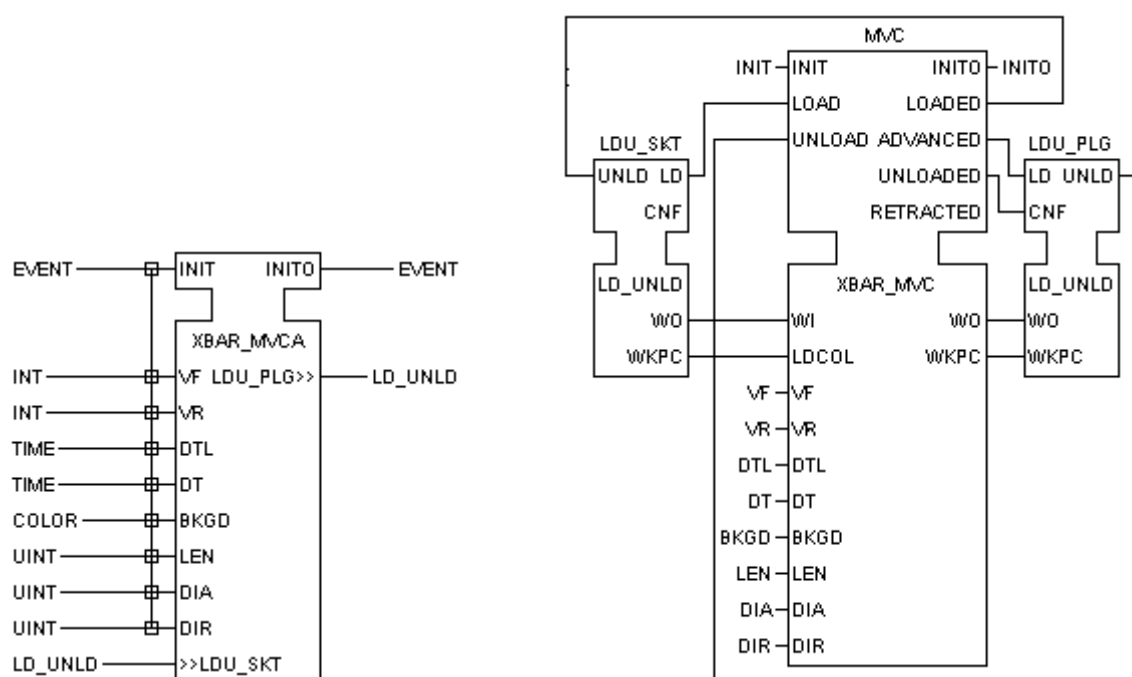


Figure 18a – Interface

Figure 18b – Body

NOTE 1 A full textual declaration of this example is given in Annex F.

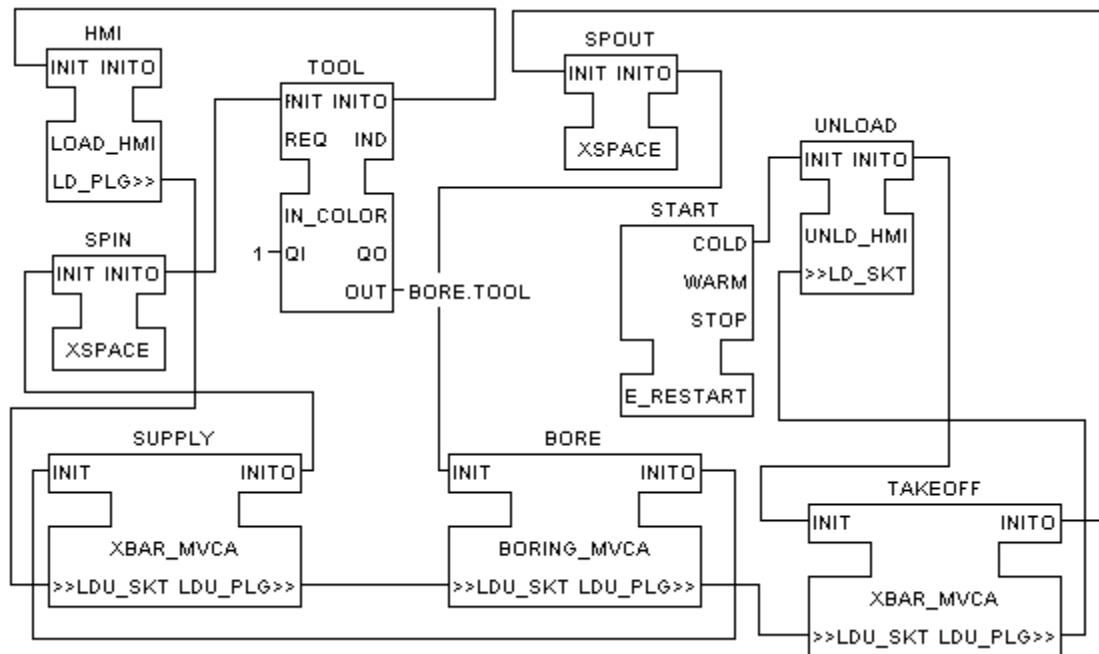
NOTE 2 This example is illustrative only. Details of the specification are not normative.

NOTE 3 Although this example presents only a composite type, *provider* and *acceptor* function block types can be either *basic* or *composite*.

Figure 18 – Illustration of provider and acceptor function block type declarations

EXAMPLE 2

Figure 19 illustrates a resource configuration containing two instances of the XBAR_MVCA type illustrated in Figure 18. The SUPPLY instance acts as an acceptor ("downstream unit") for the HMI block and a provider ("upstream unit") for the BORE block, while the TAKEOFF instance fulfills corresponding roles for the BORE and UNLOAD blocks, respectively.



NOTE 1 This example is illustrative only. Details of the specification are not normative.

NOTE 2 *Parameter* connections are omitted in this diagram for clarity.

NOTE 3 Type declarations for blocks other than the XBAR_MVCA type are not given in Annex F.

Figure 19 – Illustration of adapter connections

5.6 Exception and fault handling

Additional facilities for the prevention, recognition and handling of *exceptions* and *faults* may be provided by *resources*. Such capabilities may be modeled as *service interface function blocks*. The definition of specific function block types for prevention, recognition and handling of exceptions and faults is beyond the scope of this standard. However, *INIT*-, *CNF*- and *IND*- outputs of service interface function blocks, and the associated *STATUS* values, may be used to indicate the occurrence and type of exceptions and faults, as noted in 6.1.3.

6 Service interface function blocks

6.1 General principles

6.1.1 General

A *service interface function block* provides one or more *services* to an application, based on a *mapping* of *service primitives* to the function block's *event inputs*, *event outputs*, *data inputs* and *data outputs*.

The external interfaces of *service interface function block* types have the same general appearance as *basic function block* types. However, some inputs and outputs of service interface function block types have specialized semantics, and the behavior of *instances* of these types is defined through a specialized graphical notation for sequences of *service primitives*.

NOTE The specification of the internal operations of service interface function blocks is beyond the scope of this standard.

6.1.2 Type specification

Declaration of service interface function block types may use the standard *event inputs*, *event outputs*, *data inputs* and *data outputs* listed in Table 2, as appropriate to the particular service provided. When these are used, their semantics shall be as defined in 6.1.2. The name of the function block *type* shall indicate the provided service.

EXAMPLE Figure 20a and Figure 20b show examples of service interface function blocks in which the primary interaction is initiated by the application and by the resource, respectively.

NOTE 1 Services can provide both resource- and application-initiated interactions in the same service interface function block.

NOTE 2 Service interface types can also utilize inputs and outputs, including *plugs* and *sockets*, with names different from those given here; in such case their usage is defined in terms of appropriate sequences of service primitives.

Table 2 – Standard inputs and outputs for service interface function blocks (1 of 2)

Event inputs
INIT This event input shall be <i>mapped</i> to a <i>request primitive</i> which requests an initialization of the service provided by the function block instance, e.g., local initialization of a <i>communication connection</i> or a process interface module.
REQ This event input shall be mapped to a <i>request primitive</i> of the service provided by the function block instance.
RSP This event input shall be mapped to a <i>response primitive</i> of the service provided by the function block instance.
Event outputs
INITO This event output shall be mapped to a <i>confirm primitive</i> which indicates completion of a service initialization procedure.
CNF This event output shall be mapped to a <i>confirm primitive</i> of the service provided by the function block instance.
IND This event output shall be mapped to an <i>indication primitive</i> of the service provided by the function block instance.
Data inputs
QI: BOOL This input represents a qualifier on the <i>service primitives</i> mapped to the <i>event inputs</i> . For instance, if this input is TRUE upon the occurrence of an INIT event, initialization of the service is requested; if it is FALSE, termination of the service is requested.
PARAMS: ANY This input contains one or more <i>parameters</i> associated with the service, typically as elements of an <i>instance</i> of a <i>structured data type</i> . When this input is present, the <i>function block type</i> specification shall define its <i>data type</i> and default initial value(s). A service interface function block type specification may substitute one or more service parameter inputs for this input.
SD_1, ..., SD_m: ANY These inputs contain the data associated with <i>request</i> and <i>response primitives</i> . The <i>function block type</i> specification shall define the <i>data types</i> and default values of these inputs, and shall define their associations with event inputs in an event sequence diagram as illustrated in 6.1.3. The function block type specification may define other names for these inputs.

Table 2 (2 of 2)

Data outputs
QO: BOOL This variable represents a qualifier on the <i>service primitives</i> mapped to the <i>event outputs</i> . For instance, a TRUE value of this output upon the occurrence of an INITO event indicates successful initialization of the service; a FALSE value indicates unsuccessful initialization.
STATUS: ANY This output shall be of a <i>data type</i> appropriate to express the status of the service upon the occurrence of an event output. A service specification may indicate that the value of this output is irrelevant for some situations, for instance, for INITO+, IND+ and CNF+ as described in 6.1.3.
RD_1, ..., RD_n: ANY These outputs contain the data associated with <i>confirm</i> and <i>indication</i> primitives. The function block <i>type specification</i> shall define the <i>data types</i> and initial values of these outputs, and shall define their associations with event outputs in an event sequence diagram as described in 6.1.3. The function block type specification may define other names for these outputs.

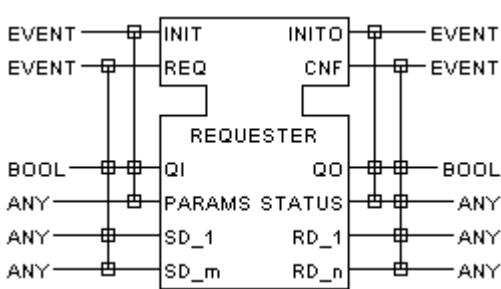


Figure 20a – Application-initiated interactions

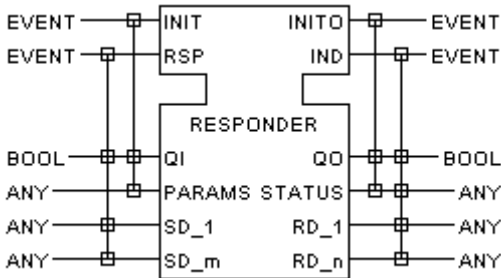


Figure 20b – Resource-initiated interactions

NOTE 1 REQUESTER and RESPONDER represent the particular services provided by instances of the function block types.

NOTE 2 The *data types* of the SD_1, ..., SD_n inputs and RD_1, ..., RD_m outputs will typically be fixed as some non-generic data type, for instance INT or WORD, in concrete implementations of the generic function block types illustrated here.

NOTE 3 See Annex F for a full textual declaration of the REQUESTER function block type.

Figure 20 – Example service interface function blocks

6.1.3 Behavior of instances

The behavior of *instances* of *service interface function blocks* shall be defined in the corresponding *function block type specification*, which can utilize *service sequence diagrams* subject to the following rules:

- a) The following semantics shall apply:
 - 1) Time increases in the downward direction.
 - 2) Events which are sequentially related are linked together across or within resources.
 - 3) If there is no specific relationship between events, in that it is impossible to foresee which will occur first but both shall occur within a finite period of time, a tilde (~) or similar textual notation is used.

- b) In the case where the service is represented by a single service interface function block, the diagram shall be partitioned by a single vertical line into two fields as illustrated in Figure 21:
 - 1) In the case where the service is provided primarily by an application-initiated interaction, the *application* shall be in the left-hand field and the *resource* in the right-hand field, as illustrated in Figure 21a.
 - 2) In the case where the service is provided primarily by a resource-initiated interaction, the *resource* shall be in the left-hand field and the *application* in the right-hand field, as illustrated in Figure 21b.
- c) In the case where the service is represented by two or more service interface function blocks, the notation illustrated in E.2.2 and E.2.3 can be used.
- d) *Service primitives* shall be indicated by horizontal arrows. The name of the *event* representing the service primitive shall be written adjacent to the arrow, and means shall be provided to determine the names of the input and/or output *variables* representing the *data* associated with the primitive.
- e) When a QI input is present in the function block type definition, the suffix "+" shall be used in conjunction with an *event input* name to indicate that the value of the QI input is TRUE upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is FALSE.
- f) When a QO output is present in the function block type definition, the suffix "+" shall be used in conjunction with an *event output* name to indicate that the value of the QO output is TRUE upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is FALSE.
- g) The standard semantics of asserted (+) and negated (-) events shall be as specified in Table 3.

Figure 21 illustrates normal sequences of service initiation, data transfer, and service termination. *Service interface function block type* specifications can utilize similar diagrams to specify all relevant sequences of service primitives and their associated data under both normal and abnormal conditions.

NOTE Sequence diagrams can also be used to document the externally observable behaviors of basic and composite function block types.

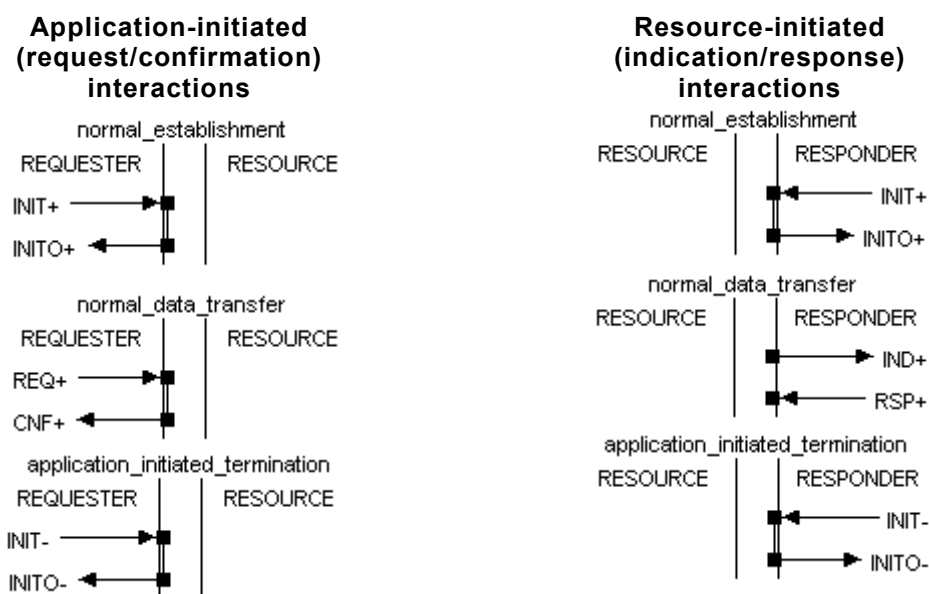


Figure 21 – Example service sequence diagrams

Table 3 – Service primitive semantics

Primitive	Semantics
INIT+	Request for service establishment
INIT-	Request for service termination
INITO+	Indication of establishment of normal service
INITO-	Rejection of service establishment request or indication of service termination
REQ+	Normal request for service
REQ-	Disabled request for service
CNF+	Normal confirmation of service
CNF-	Indication of abnormal service condition
IND+	Indication of normal service arrival
IND-	Indication of abnormal service condition
RSP+	Normal response by application
RSP-	Abnormal response by application

6.2 Communication function blocks

6.2.1 Type specification

Communication function blocks provide *interfaces* between *applications* and the "communication mapping" functions of *resources* as defined in 4.3; hence, they are *service interface function blocks* as described in 6.1.

Like other service interface function blocks, a communication function block may be of either *basic* or *composite* type, as long its operation can be represented by a *mapping* of *service primitives* to the function block's *event inputs*, *event outputs*, *data inputs* and *data outputs*.

This subclause provides rules for the *declaration* of *communication function block types*. 6.2.2 provides rules for the behavior of *instances* of such function block types. Clause E.2 defines generic communication function block types for *unidirectional* and *bidirectional transactions*, and gives rules for the implementation-dependent customization of these types.

Declaration of communication function block types shall utilize the means defined in 6.1 for the declaration of *service interface function block types*, with the specialized semantics shown in Table 4 for *input* and *output variables*.

Table 4 – Variable semantics for communication function blocks

Variable	Semantics
PARAMS	This input provides <i>parameters</i> of the <i>communication connection</i> associated with the <i>communication function block instance</i> . This shall include means of identifying the communication protocol and communication connection, and may include other parameters of the communication connection such as timing constraints, etc.
SD_1, . . . , SD_m	These inputs represent <i>data</i> to be transferred along the <i>communication connection</i> specified by the PARAMS input upon the occurrence of a REQ+ or RSP+ <i>primitive</i> , as appropriate. ^a
STATUS	This output represents the status of the <i>communication connection</i> , for instance: - Normal completion of initiation, termination, or data transfer - Reasons for abnormal initiation, termination, or data transfer
RD_1, . . . , RD_n	These outputs represent <i>data</i> received along the <i>communication connection</i> specified by the PARAMS input upon the occurrence of an IND+ or CNF+ <i>primitive</i> , as appropriate. ^a
NOTE Communication function block type declarations can define constraints between RD_1, . . . , RD_n outputs and the SD_1, . . . , SD_m inputs of corresponding function block instances. For example, the number and types of the RD outputs might be constrained to match the number and types of the corresponding SD inputs.	
^a <i>Communication function block type declarations</i> define the number and type of the SD_1, . . . , SD_m inputs and RD_1, . . . , RD_n outputs, and can assign them other names.	

6.2.2 Behavior of instances

As illustrated in Clause E.2, the behavior of *instances* of *communication function block types* shall be defined in the corresponding communication function block type *declaration*, utilizing the means specified for *service interface function blocks* in 6.1 with the specialized service primitive semantics given in Table 5. Such specification shall include *service primitive* sequences for:

- normal and abnormal establishment and release of *communication connections*;
- normal and abnormal data transfer.

Table 5 – Service primitive semantics for communication function blocks

Primitive	Semantics
INIT+	Request for communication connection establishment
INIT-	Request for communication connection release
INITO+	Indication of communication connection establishment
INITO-	Rejection of communication connection establishment request or indication of communication connection release
REQ+	Normal request for data transfer
REQ-	Disabled request for data transfer
CNF+	Normal confirmation of data transfer
CNF-	Indication of abnormal data transfer
IND+	Indication of normal data arrival
IND-	Indication of abnormal data arrival
RSP+	Normal response by application to data arrival
RSP-	Abnormal response by application to data arrival

6.3 Management function blocks

6.3.1 Requirements

Extending the functional requirements for "application management" in subclause 8.3.2 of ISO/IEC 7498-1:1994 to the distributed application model of this standard indicates that *services* for management of resources and applications in IPMCSs should be able to perform the following *functions*:

- a) In a *resource*, create, initialize, start, stop, delete, query the existence and *attributes* of, and provide notification of changes in availability and status of:
 - 1) data types
 - 2) function block types and instances
 - 3) connections among function block instances
- b) In a *device*, create, initialize, start, stop, delete, query the existence and *attributes* of, and provide notification of changes in availability and status of *resources*.

NOTE 1 The provisions of this standard are not intended to meet the requirements for *system management* addressed in ISO/IEC 7498-4 and ISO/IEC 10040, except as such requirements are addressed by the above listed functions.

NOTE 2 This standard only deals with item a) above, i.e., the management of *applications* in *resources*. A framework for device management is described in IEC 61499-2.

NOTE 3 The associations among *resources*, *applications*, and *function block instances* are defined in *system configurations* as described in 7.3.

NOTE 4 Starting and termination of a distributed *application* is performed by an appropriate *software tool*.

6.3.2 Type specification

Figure 22 illustrates the general form of *management function block types* whose *instances* meet the application management requirements defined above.

NOTE 1 In particular implementations, the type name (MANAGER in this example) might represent the type of the managed resource.

NOTE 2 For these function block types, the specific CMD and OBJECT inputs and RESULT output replace the generic SD_1 and SD_2 inputs and RD_1 output described in 6.1.

NOTE 3 The INIT and PARAMS inputs and INITO output might or might not be present in a particular implementation.

NOTE 4 When present, the type and values of the PARAMS input are **implementation-dependent** parameters of the resource type.

NOTE 5 A full textual specification of this function block type, including all service sequences, is given in Annex F.

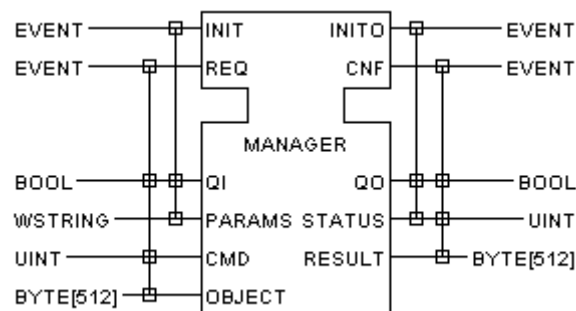
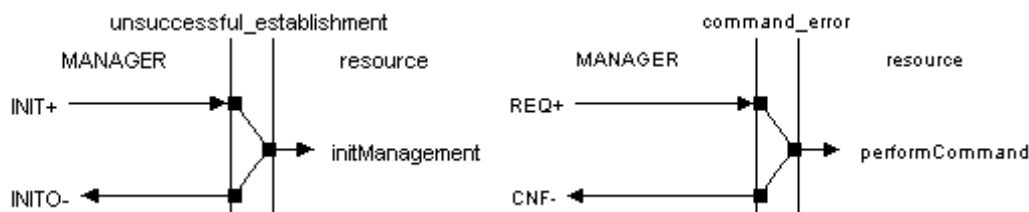


Figure 22 – Generic management function block type

The behavior of instances and input/output semantics of management function block types shall follow the rules given in 6.1 for *service interface function block types* with application

initiated interactions, with the additional behaviors shown in Figure 23 for unsuccessful service initiation and requests.



NOTE A full textual specification of this function block type, including all service sequences, is given in Annex F.

Figure 23 – Service primitive sequences for unsuccessful service

The management *operation* to be *executed* shall be expressed by the value of the `CMD` input of a management function block according to the semantics defined in Table 6.

Table 6 – CMD input values and semantics

Value	Command	Semantics
0	CREATE	Create specified object
1	DELETE	Delete specified object
2	START	Start specified object
3	STOP	Stop specified object
4	READ	Read parameter data
5	WRITE	Write parameter data
6	KILL	Make specified object unrunnable
7	QUERY	Request information on specified object
8	RESET	Reset specified object

The values and corresponding semantics of the `STATUS` output of a management function block shall be as described in Table 7 to express the result of performing the specified command.

Table 7 – STATUS output values and semantics

Value	Status	Semantics
0	RDY	No errors
1	BAD_PARAMS	Invalid PARAMS input value
2	LOCAL_TERMINATION	Application-initiated termination
3	SYSTEM_TERMINATION	System-initiated termination
4	NOT_READY	Manager is not able to process the command
5	UNSUPPORTED_CMD	Requested command is not supported
6	UNSUPPORTED_TYPE	Requested object type is not supported
7	NO_SUCH_OBJECT	Referenced object does not exist
8	INVALID_OBJECT	Invalid object specification syntax
9	INVALID_OPERATION	Commanded operation is invalid for specified object
10	INVALID_STATE	Commanded operation is invalid for current object state
11	OVERFLOW	Previous transaction still pending