BS EN 61499-1:2013

BSI Standards Publication

Function blocks

Part 1: Architecture

bsi.

...making excellence a habit.™

National foreword

This British Standard is the UK implementation of EN 61499-1:2013. It is identical to IEC 61499-1:2012. It supersedes BS EN 61499-1:2005 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee GEL/65, Measurement and control.

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

Published by BSI Standards Limited 2013

ISBN 978 0 580 78490 3

ICS 25.040.40; 35.240.50

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 April 2013.

Amendments issued since publication

Amd. No.	Date	Text affected

EUROPEAN STANDARD

NORME EUROPÉENNE

EUROPÄISCHE NORM

EN 61499-1

February 2013

ICS 25.040; 35.240.50

Supersedes EN 61499-1:2005

English version

Function blocks - Part 1: Architecture (IEC 61499-1:2012)

Blocs fonctionnels -
Partie 1: Architecture
(CEI 61499-1:2012)

Funktionsbausteine für industrielle Leitsysteme -
Teil 1: Architektur
(IEC 61499-1:2012)

This European Standard was approved by CENELEC on 2012-12-12. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.

CENELEC
European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Management Centre: Avenue Marnix 17, B - 1000 Brussels

Foreword

The text of document 65B/845/FDIS, future edition 2 of IEC 61499-1, prepared by SC 65B "Measurement and control devices" of IEC/TC 65 "Industrial-process measurement, control and automation" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN 61499-1:2013.

The following dates are fixed:

• latest date by which the document has to be implemented at national level by publication of an identical national standard or by endorsement	(dop)	2013-09-12
• latest date by which the national standards conflicting with the document have to be withdrawn	(dow)	2015-12-12

This document supersedes EN 61499-1:2005.

EN 61499-1:2013 includes the following significant technical changes with respect to EN 61499-1:2005:

- *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*.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC [and/or CEN] shall not be held responsible for identifying any or all such patent rights.

Endorsement notice

The text of the International Standard IEC 61499-1:2012 was approved by CENELEC as a European Standard without any modification.

In the official version, for Bibliography, the following notes have to be added for the standards indicated:

IEC 61131-5:2000	NOTE	Harmonised as EN 61131-5:2001 (not modified).
IEC 61499 Series	NOTE	Harmonised as EN 61499 Series (not modified).
IEC 61499-2:2012	NOTE	Harmonised as EN 61499-2:2013 (not modified).
IEC 61499-4	NOTE	Harmonised as EN 61499-4.

Annex ZA
(normative)

Normative references to international publications with their corresponding European publications

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.

NOTE When an international publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

Publication	Year	Title	EN/HD	Year
IEC 61131-1	-	Programmable controllers - Part 1: General information	EN 61131-1	-
IEC 61131-3	2003	Programmable controllers - Part 3: Programming languages	EN 61131-3	2003
ISO/IEC 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)	-	-

CONTENTS

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

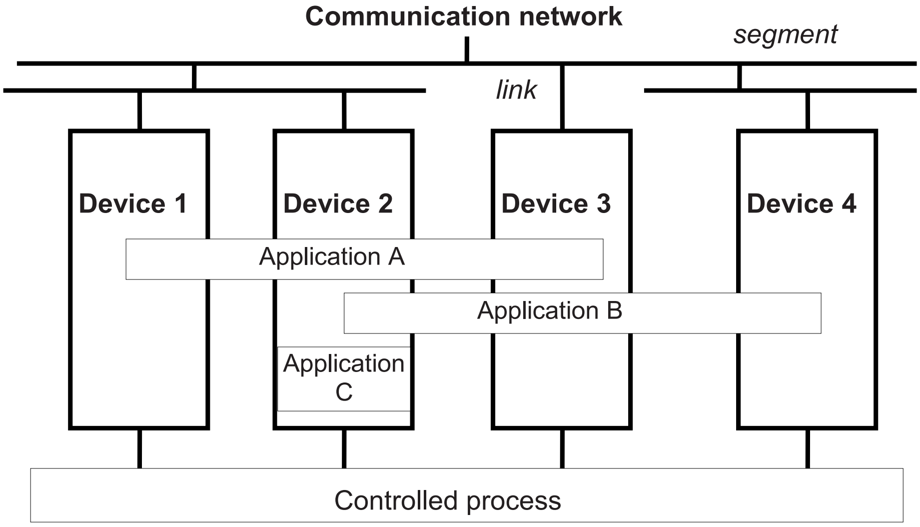
Note 1 to entry: The values of a variable are usually restricted to a certain *data type*.

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.

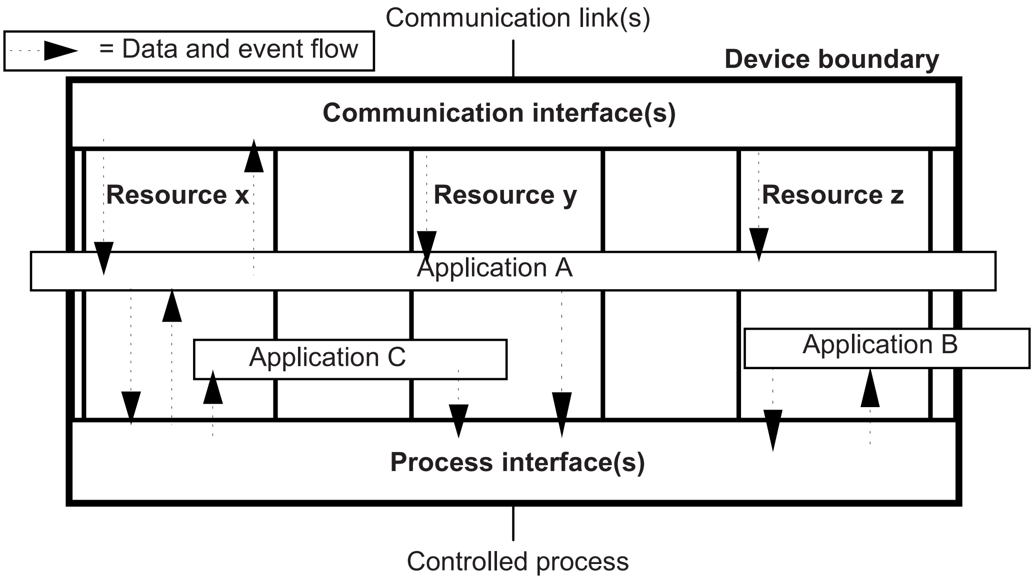
NOTE 2 A device that contains no resources is considered to be functionally equivalent to a *resource* as defined in 4.3.

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.

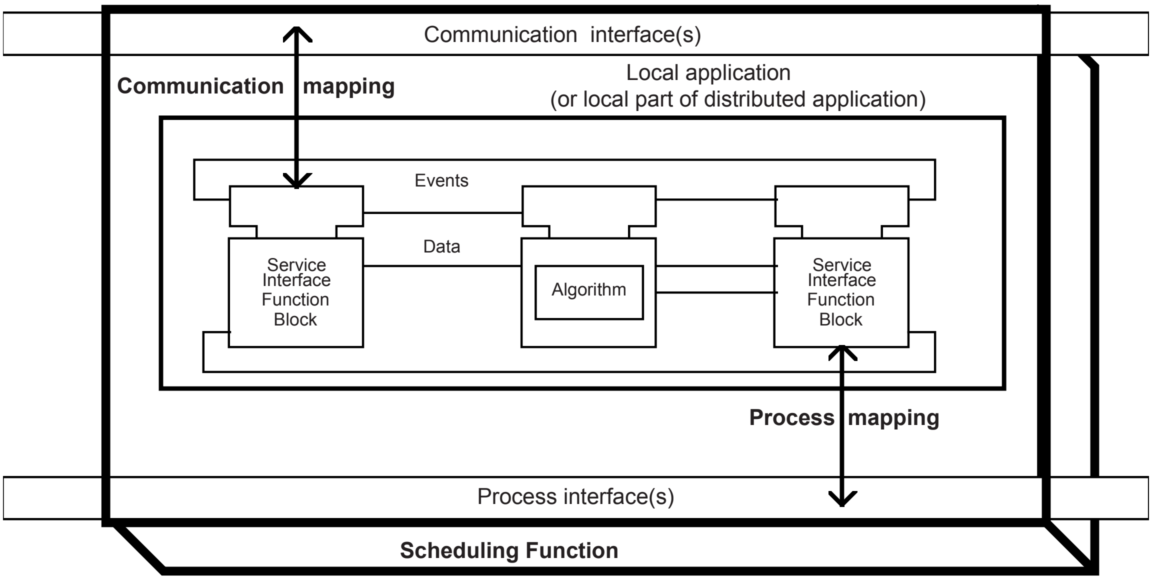
The *functions* of a resource are to accept *data* and/or *events* from the process and/or communication *interfaces*, process the data and/or events, and to return data and/or events to the process and/or communication interfaces, as specified by the *applications* utilizing the resource.

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*.

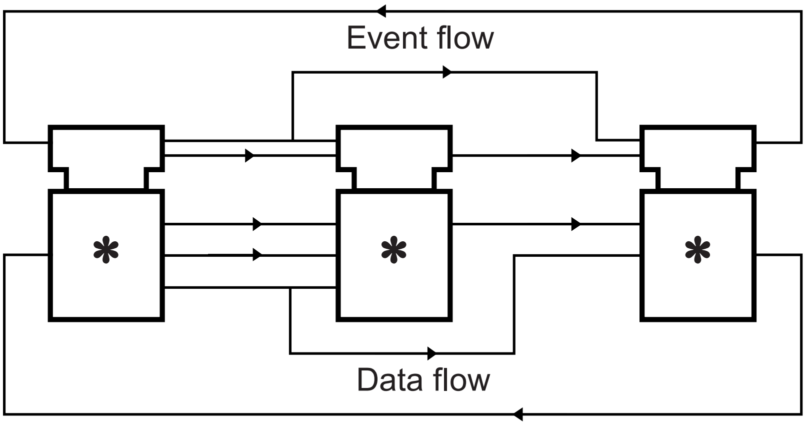
Subapplications are *instances* of *subapplication types*, which like applications consist of *function block networks*. Application names, subapplication and function block *instance names* may therefore be used to create a hierarchy of *identifiers* that can uniquely identify every *function block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

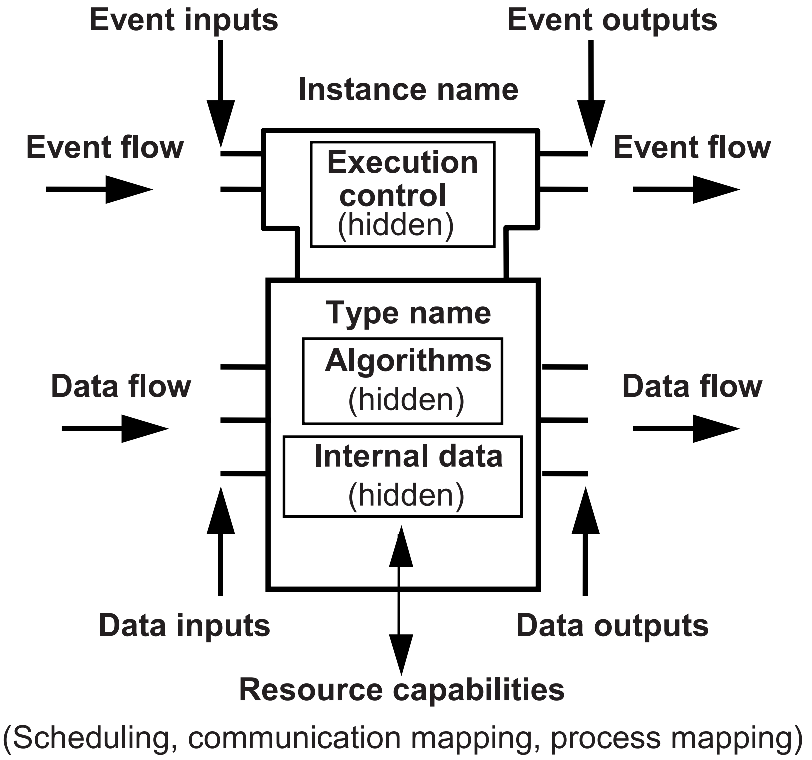
4.5.1 Characteristics of function block instances

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.

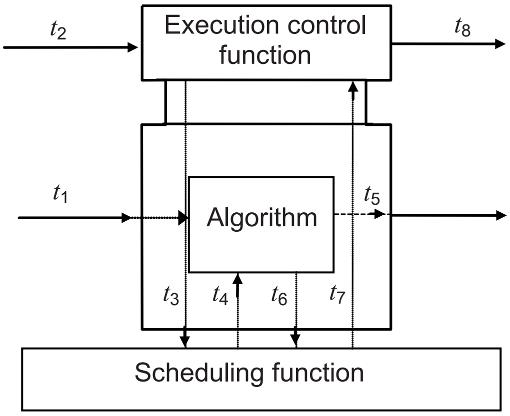
As shown in Figure 7, the significant timing delays in this case which are of interest in application design are:

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

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

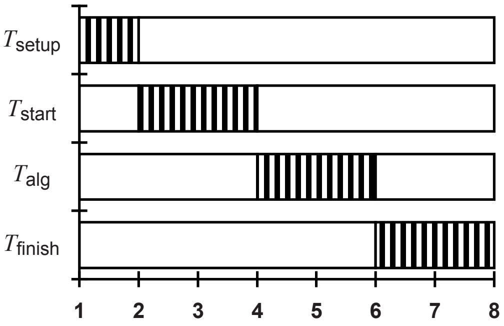
$T_{\text{alg}} = t_6 - t_4$ (algorithm execution time)

$T_{\text{finish}} = t_8 - t_6$ (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,... 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*.

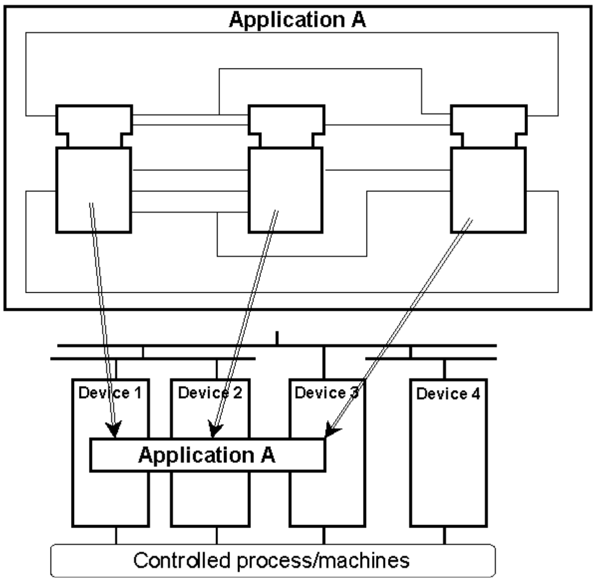


Figure 8a – Distribution model

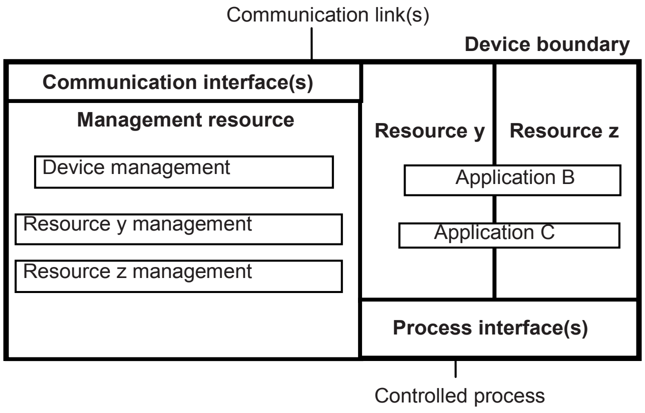


Figure 8b – Shared management model

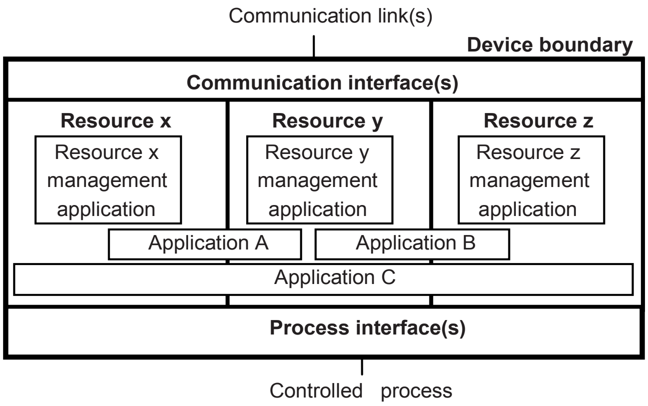


Figure 8c – Distributed management model

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.

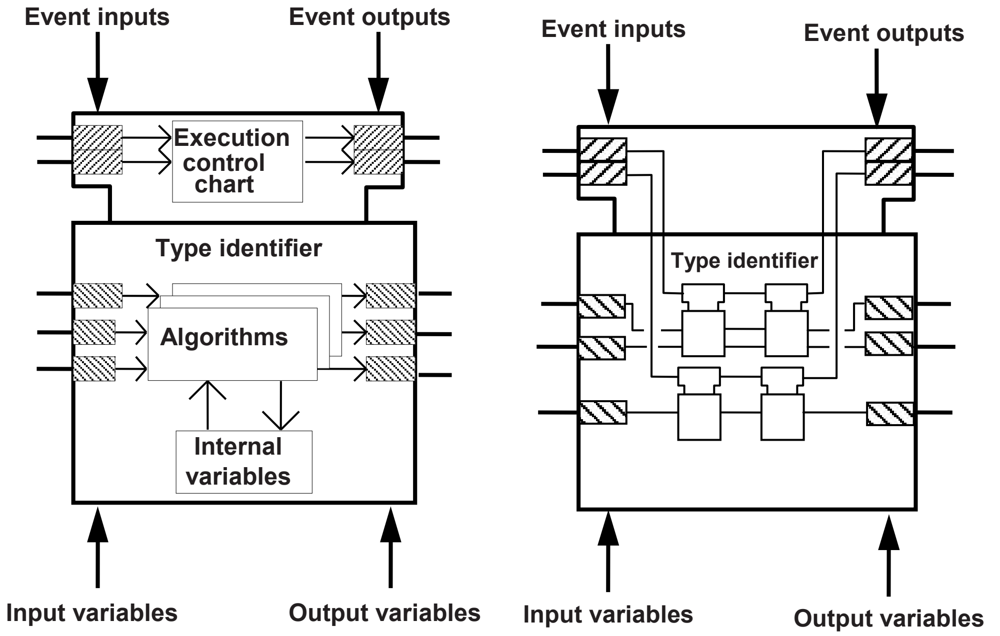
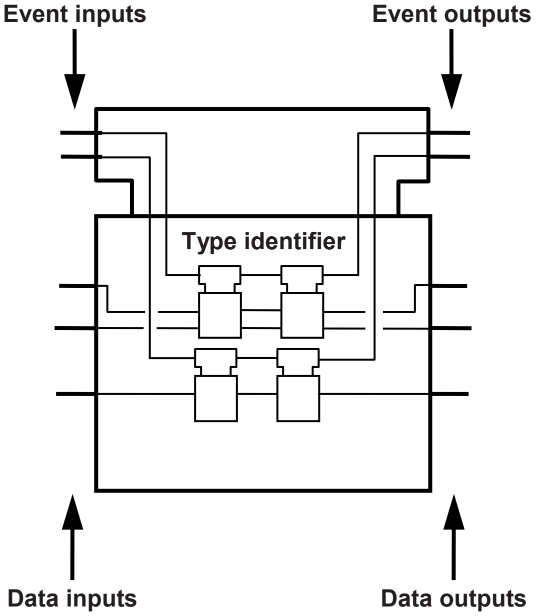


Figure 9a – Basic function block (5.2)

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:

a) the function block *type name* is shown at the top center of the lower portion of the block;

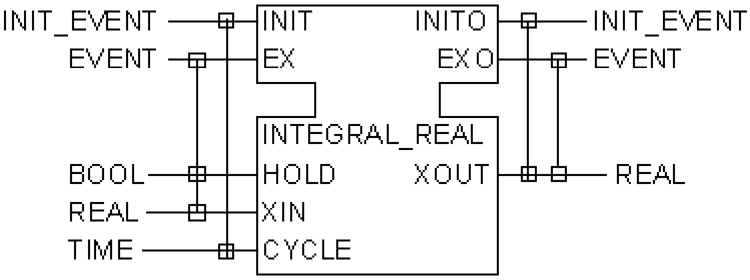
b) the names and *type declarations* of *input variables* and *socket adapters* are shown at the left edge of the lower portion of the block;

c) the names and *type declarations* of *output variables* and *plug adapters* are shown at the right edge of the lower portion of the block;

d) 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;

e) the *algorithms* associated with the function block type are declared as specified in 5.2.1.3;

f) 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.

a) *Event interfaces* are located in a distinct area at the top of the block.

b) *Event input* names are shown at the left-hand side of the upper portion of the block.

c) *Event output* names are shown at the right-hand side of the upper portion of the block.

d) 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.

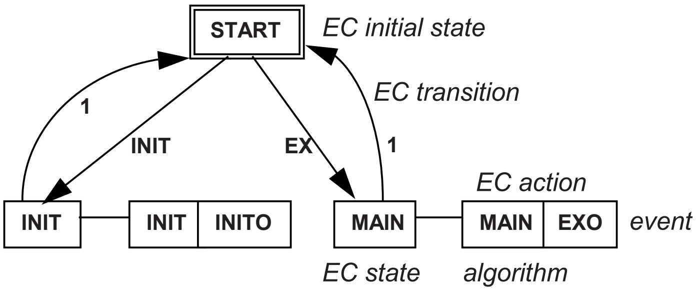


Figure 11 – ECC example

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:

a) 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.

b) 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.

c) 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.

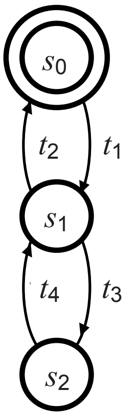


Figure 12 – ECC operation state machine

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.

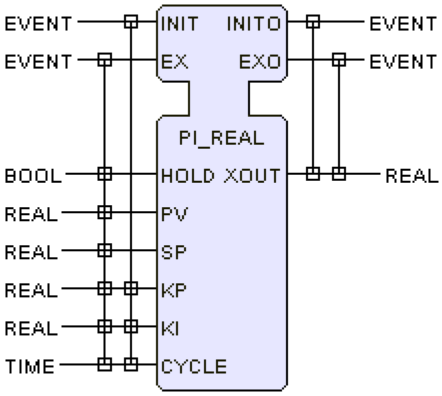


Figure 13a – External interface

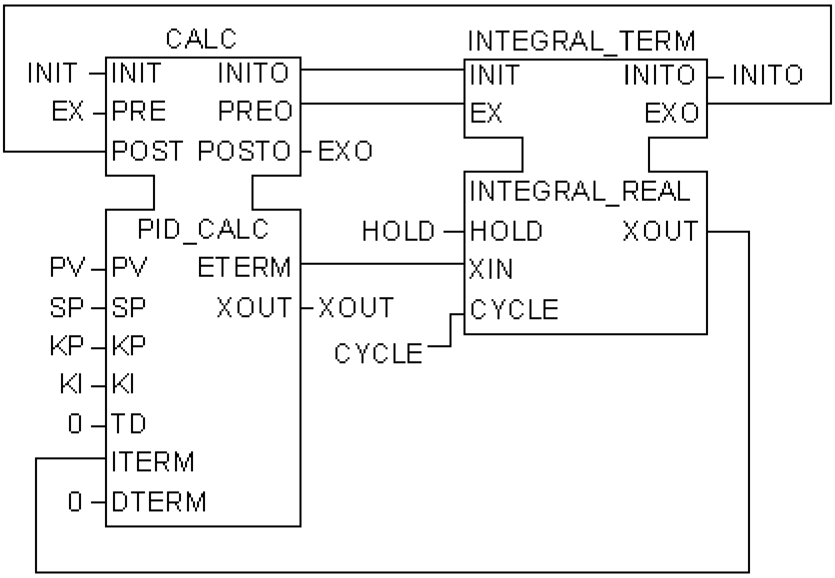


Figure 13b – Graphical body

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

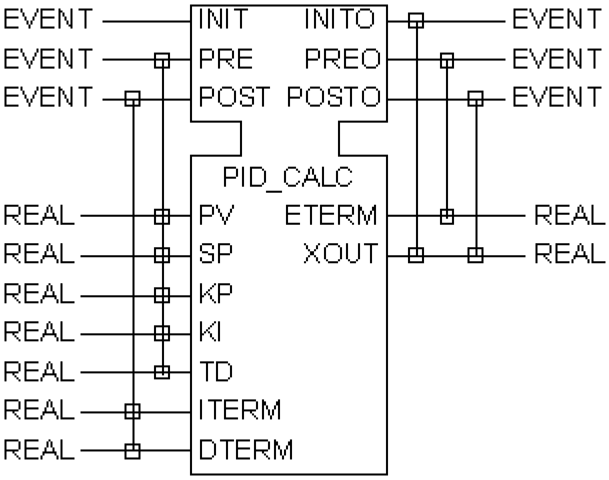


Figure 14a – External interface

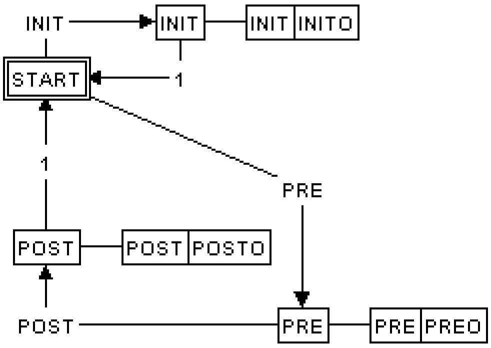


Figure 14b – Execution control

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.

a) 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.

b) 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.

c) 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.

d) 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.

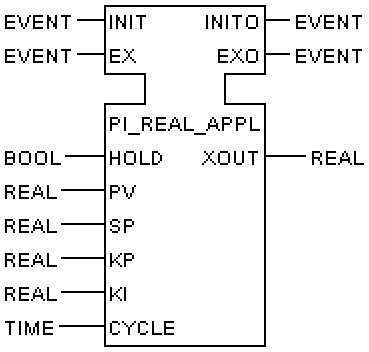


Figure 15a – External interface

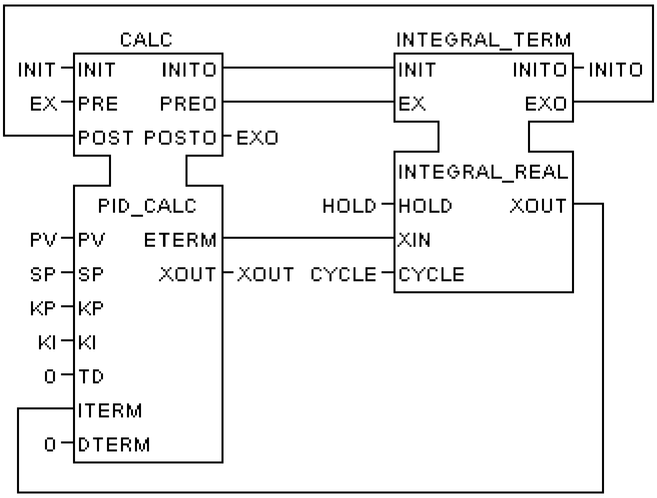


Figure 15b – Graphical body

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:

a) 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.

b) 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.

c) 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.

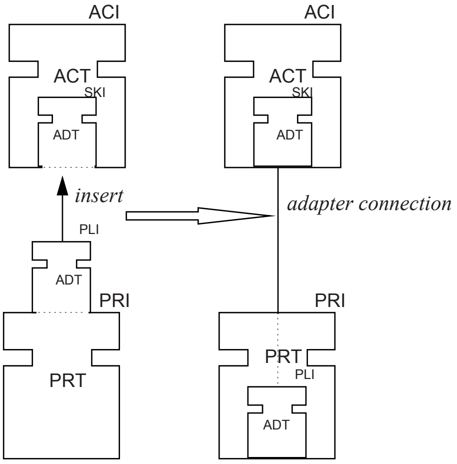
Since subapplications do not explicitly create variables, no specific initialization procedures are applicable to subapplication instances.

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:

a) 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.

b) 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.

c) 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.

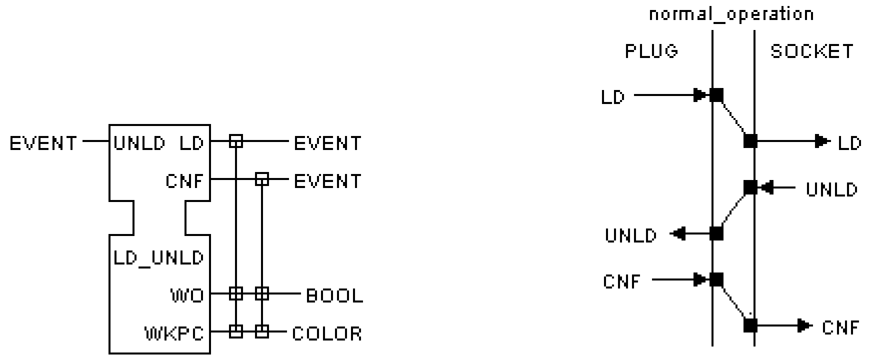


Figure 17a – Interface

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:

a) 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.

b) 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.

c) *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.

d) *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.

e) 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.

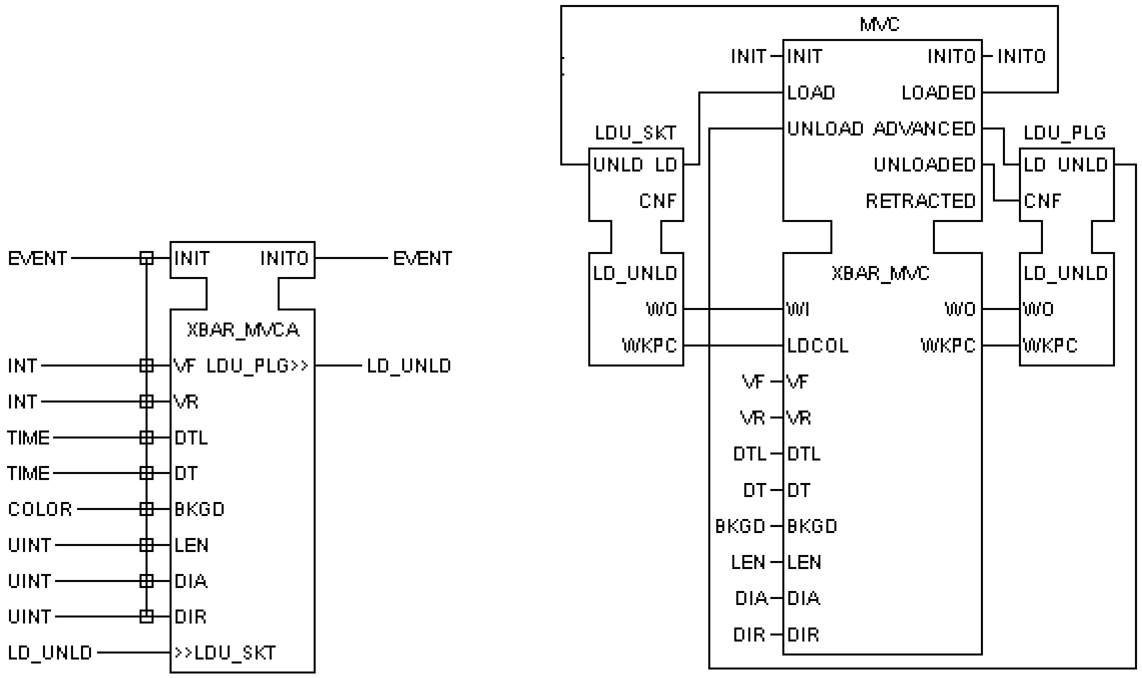


Figure 18a – Interface **Figure 18b – Body**

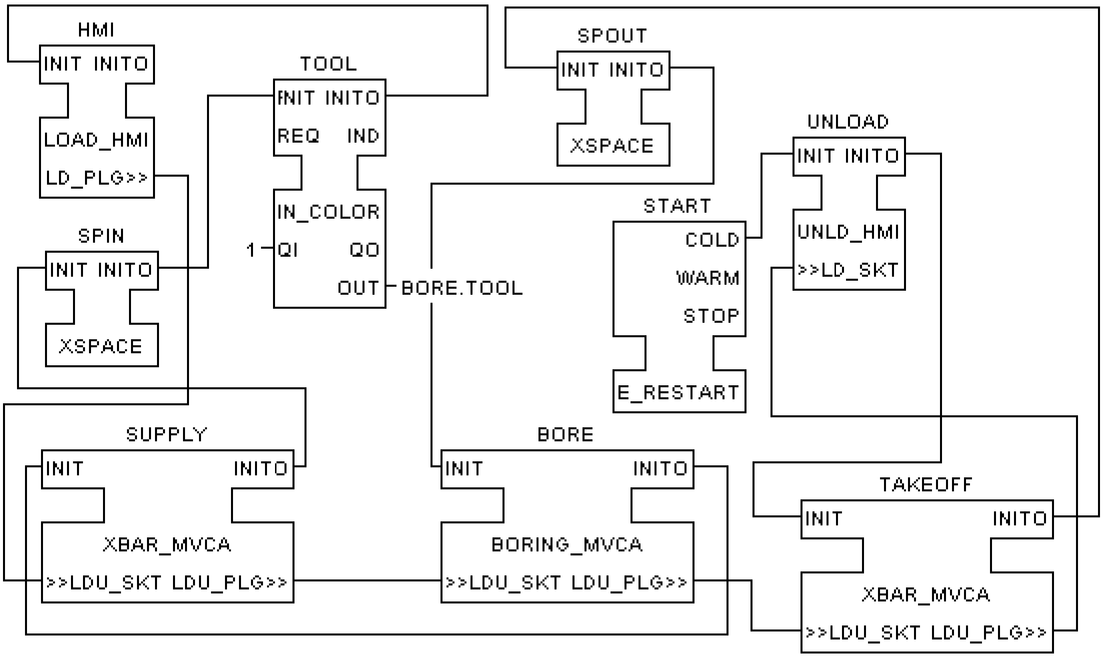
NOTE 1 A full textual declaration of this example is given in Annex F.

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 *mapped* to a *request primitive* which requests an initialization of the service provided by the function block instance, e.g., local initialization of a *communication connection* or a process interface module.
`REQ` This event input shall be mapped to a *request primitive* of the service provided by the function block instance.
`RSP` This event input shall be mapped to a *response primitive* of the service provided by the function block instance.
Event outputs
`INITO` This event output shall be mapped to a *confirm primitive* which indicates completion of a service initialization procedure.
`CNF` This event output shall be mapped to a *confirm primitive* of the service provided by the function block instance.
`IND` This event output shall be mapped to an *indication primitive* of the service provided by the function block instance.
Data inputs
`QI`: **`BOOL`** This input represents a qualifier on the *service primitives* mapped to the *event inputs*. 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 *parameters* associated with the service, typically as elements of an *instance* of a *structured data type.* When this input is present, the *function block type* specification shall define its *data type* 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 *request* and *response primitives*. The *function block type* specification shall define the *data types* 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 *service primitives* mapped to the *event outputs*. 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 *data type* 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 *confirm* and *indication primitives*. The function block *type specification* shall define the *data types* 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.

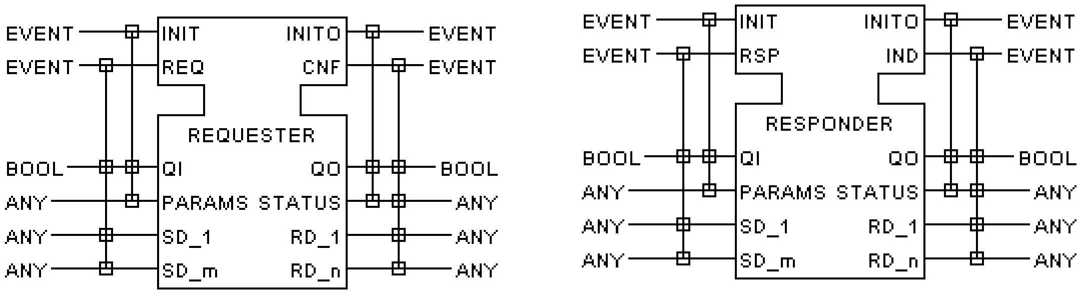


Figure 20a – Application-initiated interactions **Figure 20b – Resource-initiated interactions**

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.

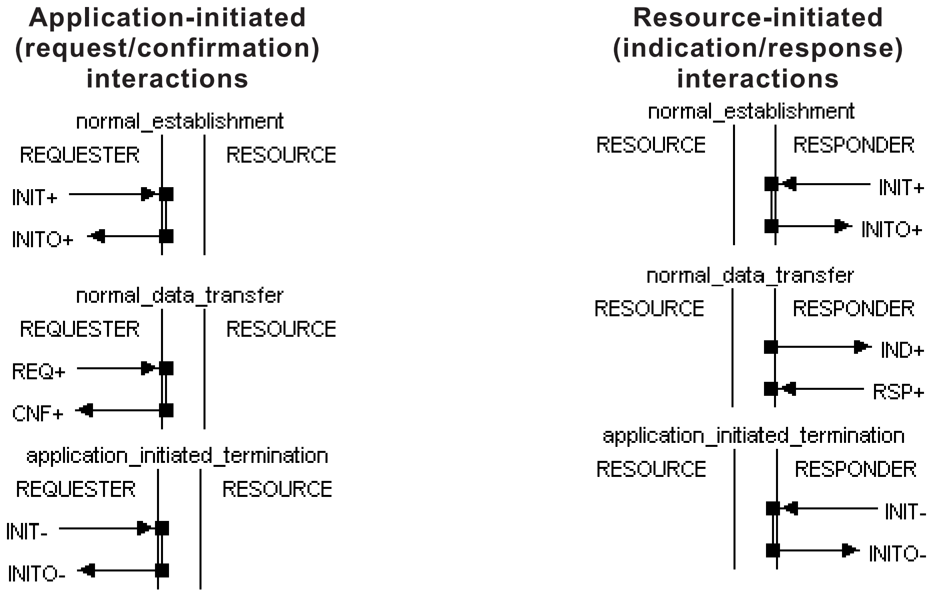
e) When a `QI` input is present in the function block type definition, the suffix "+" shall be used in conjunction with an *event input* name to indicate that the value of the `QI` input is `TRUE` upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is `FALSE`.

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.



Application-initiated (request/confirmation) interactions

Resource-initiated (indication/response) interactions

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 *parameters* of the *communication connection* associated with the *communication function block instance*. 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 *data* to be transferred along the *communication connection* specified by the `PARAMS` input upon the occurrence of a `REQ+` or `RSP+` *primitive*, as appropriate. [a]
STATUS	This output represents the status of the *communication connection*, 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 *data* received along the *communication connection* specified by the `PARAMS` input upon the occurrence of an `IND+` or `CNF+` primitive, 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] *Communication function block type declarations* 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.

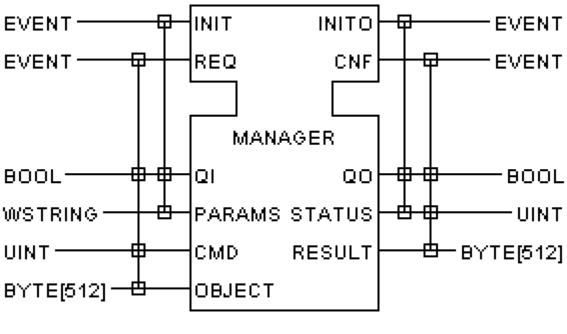
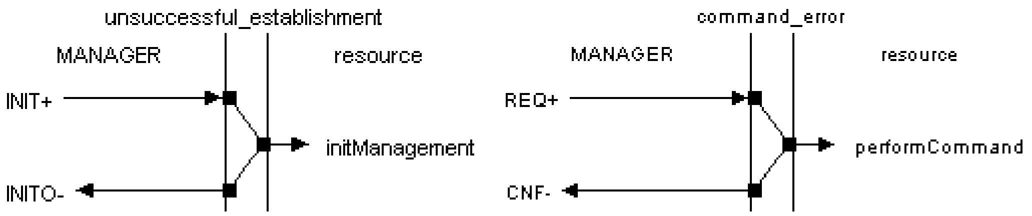


Figure 22 – Generic management function block type

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

The actual lengths of the `OBJECT` input and `RESULT` output of management function block instances are **implementation-dependent**.

The `OBJECT` input shall specify the object to be operated on according to the `CMD` input, and the `RESULT` output shall contain a description of the object resulting from the operation if successful. The contents of these strings shall consist of **implementation-dependent** encodings of objects defined as non-terminal symbols in Annex B and referenced in Table 8.

NOTE 6 The maximum allowable length of the `OBJECT` input and `RESULT` output is an **implementation-dependent parameter**; the value of 512 given in Figure 22 is illustrative.

Table 8 – Command syntax

CMD	OBJECT	RESULT
CREATE	type_declaration	data_type_name
	fb_type_declaration	fb_type_name
	fb_instance_definition	fb_instance_reference
	connection_definition	connection_start_point
DELETE	data_type_name	data_type_name
	fb_type_name	fb_type_name
	fb_instance_reference	fb_instance_reference
	connection_definition	connection_definition
START	fb_instance_reference	fb_instance_reference
	application_name	application_name
STOP	fb_instance_reference	fb_instance_reference
	application_name	application_name
KILL	fb_instance_reference	fb_instance_reference
QUERY	all_data_types	data_type_list
	all_fb_types	fb_type_list
	data_type_name	type_declaration
	fb_type_name	fb_type_declaration
	fb_instance_reference	fb_status
	connection_start_point	connection_end_points
	application_name	fb_instance_list
READ	parameter_reference	parameter
WRITE	referenced_parameter	parameter_reference
RESET	fb_instance_reference	fb_status
NOTE See Table 6 for the integer values of the `CMD` input corresponding to the commands listed above.		

It shall be an **error**, resulting in a `STATUS` code of `INVALID_OBJECT`, if a `CREATE` command attempts to create

- a *function block* whose *instance name* duplicates that of an existing function block within the same *resource*,
- a duplicate *connection*, or
- multiple connections to a *data input.*

The single exception to the above rule is that a `CREATE` command can replace a connection of a *parameter* to a *data input* with a new parameter connection.

It shall be an **error**, resulting in a `STATUS` code of `UNSUPPORTED_TYPE`, if a `CREATE` command attempts to create a function block instance or parameter of a *type* which is not known to the management function block.

It shall be an **error**, resulting in a `STATUS` code of `INVALID_OPERATION`, if a `DELETE` command attempts to delete a *function block type*, function block instance, *data type* or connection which is defined in the *type specification* of the managed *resource*.

The semantics of the `START` and `STOP` commands shall be as follows:

- `START` and `STOP` of a *function block instance* shall be as defined in 6.3.2;
- `START` and `STOP` of an *application* shall be equivalent to `START` and `STOP`, respectively, of all *function block instances* in the application contained within the managed *resource*;
- `STOP` of a *management function block instance* shall be equivalent to `STOP` of all *function block instances* within the managed *resource*;
- `START` of a *management function block instance* shall be equivalent to `START` of all *function block instances* within the managed *resource.* If the managed resource was previously stopped, this shall be followed by issuing of an event at the appropriate output of each instance of the `E_RESTART` function block type defined in Annex A*.* These events shall occur at the `WARM` outputs of the `E_RESTART` blocks if the resource was stopped due to a previous `STOP` command, and at the `COLD` outputs otherwise.

Specialized semantics for the `QUERY` command shall be as follows:

- when the `OBJECT` input specifies an *event input, event output* or *data output,* the `RESULT` output shall contain zero or more opposite end points;
- when the `OBJECT` input specifies a *data input,* the `RESULT` output shall list zero or one opposite end point;
- when the `OBJECT` input specifies the name of an *application,* the `RESULT` output shall list the names of all function blocks in the application contained within the managed *resource*.

6.3.3 Behavior of managed function blocks

Function blocks that are under the control of a *management function block* shall exhibit operational behaviors equivalent to that shown in the state transition diagram of Figure 24, subject to the following rules.

a) The capitalized transition conditions in Figure 24 refer to a value of the `CMD` input, as specified in Table 6*,* of the management function block upon the occurrence of a `REQ+` service primitive.

b) The `command_error` sequence of primitives for the `MANAGER` function block type shall occur, with the indicated value of the `STATUS` output as defined in Table 7*,* under the following conditions:

   1) `UNSUPPORTED_CMD`: No state exists in Figure 24 with a transition condition for the specified `CMD` value;

   2) `INVALID_STATE`: The currently active state does not have a transition condition for the specified `CMD` value;

   3) `UNSUPPORTED_TYPE`: The `CMD` value is `CREATE`, and the function block instance does not exist, but the function block type is unknown to the `MANAGER` instance, i.e., the guard condition **`type_defined`** is `FALSE`;

   4) `INVALID_OPERATION`: The `CMD` value is `DELETE`, and the function block instance is in the `STOPPED` or `KILLED` state, but the function block instance is *declared* in the *device* or *resource type* specification, i.e., the guard condition **`is_deletable`** is `FALSE`.

c) The `normal_command_sequence` of primitives shown for the `MANAGER` function block type shall follow a `CMD+` service primitive under all other conditions, with a value of `RDY` for the `STATUS` output as defined in Table 7, and a corresponding value for the `RESULT` output as defined in Table 8.

d) The semantics of the actions shown in Figure 24 shall be as shown in Table 9 for managed *basic* and *service interface function blocks*.

e) The actions described in the previous rule apply recursively to all *component function blocks* of managed *composite function blocks*.

NOTE 1 The behaviors of function blocks that are not under the control of management function blocks are beyond the scope of this standard.

NOTE 2 Specification of the behavior of managed function blocks under conditions of power loss and restoration is beyond the scope of this standard. Such behavior can be specified by the manufacturer of a compliant device, for example by reference to an appropriate standard.

NOTE 3 *Applications* can utilize *instances* of the `E_RESTART` block described in Annex A to generate events that can be used to trigger appropriate algorithms upon power loss and restoration.

NOTE 4 As described in 5.4.2, execution control in *subapplications* is entirely deferred to the execution control mechanisms of their component function blocks and component subapplications.

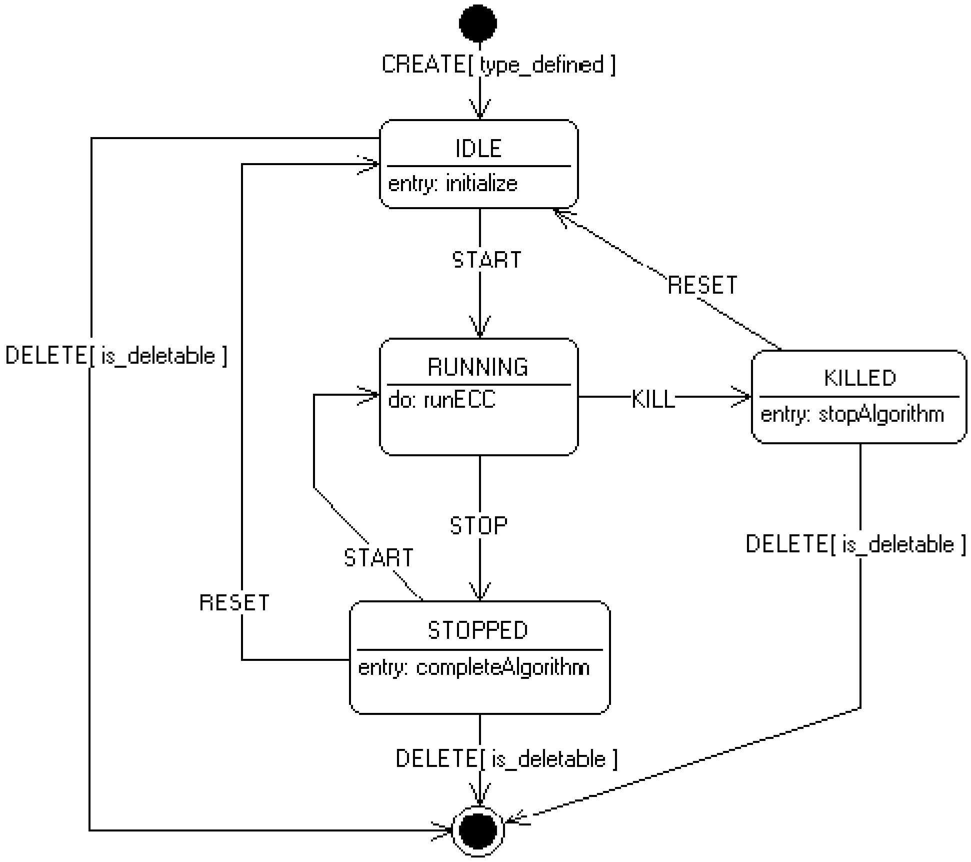


Figure 24 – Operational state machine of a managed function block

Table 9 – Semantics of actions in Figure 24

Action	Basic function blocks	Service interface function block
initialize	Initialize all variables as defined in 5.2.2.1.	
	Perform other initialization operations as defined in 5.2.2.1.	Place service in the proper state to respond correctly to an `INIT+` primitive.
runECC	Enable operation of the ECC state machine defined in 5.2.2.2.	Enable invocation of service primitives by events at event inputs, and generation of events at event outputs.
completeAlgorithm	Allow the currently active algorithm (if any) without further generation of output events.	Allow the currently active service primitive to complete.
stopAlgorithm	Terminate the operations of the currently active algorithm (if any) immediately.	Terminate all operations of the service immediately.

7 Configuration of functional units and systems

7.1 Principles of configuration

Clause 7 contains rules for the *configuration* of industrial-process measurement and control *systems* (IPMCSs) according to the following model:

a) an IPMCS consists of interconnected *devices;*

b) a *device* is an *instance* of a corresponding device *type;*

c) the functional capabilities of a *device type* are described in terms of its associated *resources;*

d) a *resource* is an *instance* of a corresponding resource *type*;

e) the functional capabilities of a *resource type* are described in terms of the *function block types* which can be *instantiated,* and the particular *function block instances* which exist, in all *instances* of the resource *type.*

The *configuration* of an IPMCS is thus considered to consist of the *configuration* of its associated *devices* and *applications,* including the allocation of *function block instances* in each *application* to the *resources* associated with the *devices*. Clause 7 defines the following sets of rules to support this process:

- rules for the functional specification of *types* of *resources* and *devices* are defined in 7.2;
- rules for the *configuration* of an IPMCS in terms of its associated *devices* and *applications* are defined in 7.3.

7.2 Functional specification of resource, device and segment types

7.2.1 Functional specification of resource types

The functional specification of a *resource type* includes:

- the resource *type name*;
- the *instance name, data type*, and initialization of each of the resource *parameters*;
- a declaration of the *data types* and *function block types* that each *instance* of the *resource type* is capable of *instantiating*;
- the instance names, types, and initial values of any function block instances that are always present in each instance of the resource type;
- any *data connections, adapter connections* and *event connections* that are always present in each instance of the resource type.

NOTE 1 Additional information can be supplied with resource type specifications, including:

- the maximum numbers of *data connections, adapter connections* and *event connections* that can exist in an instance of the resource type;
- the time (identified as T_{alg} in Figure 7) required for *execution* of each *algorithm* of function blocks of a specified type in an instance of the resource;
- the maximum number of instances of specified function block types that can exist in each instance of the resource;
- trade-offs among function block instances, e.g., whether two instances of function block type "A" can be traded for one instance of type "B", etc.

NOTE 2 The functional specifications of a resource's communication and process *interfaces*, including the kind and degree of compliance to applicable standards, is beyond the scope of this standard except as such interfaces are represented by *service interface function blocks.*

7.2.2 Functional specification of device types

The functional specification of a *device type* includes:

a) the device *type name*;
b) the *instance name, data type*, and initialization of each of the device *parameters*;
c) the instance name, type name, and initialization of each *function block instance* that is always present in each *instance* of the device type;
d) any *data connections, adapter connections* and *event connections* that are always present in each instance of the device type;
e) declarations of the *resource instances* which are present in each instance of the device type. Each such declaration shall contain:
    1) the resource instance name and type name;
    2) the instance name, type name, and initialization of each *function block instance* that is always present in the resource instance in each instance of the device type;
    3) any *data connections, adapter connections* and *event connections* that are always present in the resource instance in each instance of the device type.

NOTE 1 Items (2) and (3) above are considered to be in addition to the corresponding elements declared in the resource type specification as defined in 7.2.1.

NOTE 2 The functional specifications of a device's communication and process *interfaces*, including the kind and degree of compliance to applicable standards, is beyond the scope of this standard except as such interfaces are represented by *service interface function blocks*.

NOTE 3 A device type can contain a function block network only when it is considered to consist of a single (undeclared) resource; in such a case the device type does not contain any declarations of resource instances.

7.2.3 Functional specification of segment types

The functional specification of a *segment type* includes:

- the segment *type name*;
- the *instance name, data type*, and initialization of each of the segment *parameters*.

7.3 Configuration requirements

7.3.1 Configuration of systems

The configuration of a *system* includes:

- the *name* of the system;
- the specification of each *application* in the system, as specified in 7.3.2;
- the configuration of each *device* and its associated *resources*, as specified in 7.3.3;

- the configuration of each *network segment* and its associated *links* to devices or resources, as specified in 7.3.4.

7.3.2 Specification of applications

The specification of an *application* consists of:

- its name in the form of an *identifier*;
- the *instance name, type name, data connections, event connections* and *adapter connections* of each *function block* and *subapplication* in the application.

It shall be an **error** if the name of an application is not unique within the scope of the *system*.

7.3.3 Configuration of devices and resources

The configuration of a *device* consists of:

- the *instance name* and *type name* of the device;
- configuration-specific values for the device *parameters*;
- the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
- the *instance name* and *type name* of each *function block instance* that is present in the device instance in addition to those defined for the device type;
- any *data connections, adapter connections* and *event connections* that are present in the device instance in addition to those defined for the device type;
- the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
- the configuration of each of the *resources* in the device. These consist of any resource instances defined in the device *type* specification, plus any additional resources associated with the specific device *instance*.

NOTE A device instance can contain a function block network only when it is considered to consist of a single (undeclared) resource; in such a case the declaration of the device instance does not contain any declarations of resource instances.

It shall be an **error** if the instance name of each device is not unique within the scope of the system.

The configuration of a *resource* consists of:

a) its *instance name* and *type name*;
b) the *data types* and *function block types* supported by the resource *instance;*
c) the *instance name, type name*, and initialization of each function block instance that is present in the resource instance;
d) any *data connections, event connections* and *adapter connections* that are present in the resource instance.

Resource configuration is subject to the following rules:

– Items b), c), and d) above are considered to be in addition to the corresponding elements declared in the device and resource type specifications as defined in 7.2.2 and 7.2.1, respectively.
– Items c) and d) include *function block instances, data connections, adapter connections* and *event connections* from those portions of *applications* allocated to the resource.
– Items c) and d) include *communication function blocks, data connections, event connections* and *adapter connections* as necessary to establish and maintain the data and event flows for any associated *applications*.

- The items in Item c) may include the *mapping* of function block instances in the application to function block instances existing in the resource as a result of type definition as described in 7.2.1.
- It shall be an **error** if the instance name of a resource is not unique within the scope of the device containing it, or if any function block instance in an application is not allocated to exactly one resource.

Automated means may be provided to meet the above requirements. Providers of such means shall either provide unambiguous rules by which their operation can be determined, or shall provide means by which the results of the application of such means can be examined and modified.

7.3.4 Configuration of network segments and links

The configuration of a *network segment* consists of:

- the *instance name* and *type name* of the segment;
- configuration-specific values for the *parameters* of the network segment.

It shall be an **error** if the *instance name* of each network segment is not unique within the scope of the *system*, or if the declared values of the segment parameters are inconsistent with the declaration (if any) of the *segment type* defined in 7.2.3.

The configuration of a *link* consists of:

- the name of a *device* or the hierarchical name of a "communication *resource*" inside a device, and the name of the network segment to which the device or the resource is connected;
- configuration-specific values for the *parameters* of the link.

Annex A
(normative)

Event function blocks

Instances of the function block *types* shown in Table A.1 can be used for the generation and processing of *events* in *composite function blocks*; in *subapplications*; in the definition of *resource* and *device types*; and in the *configuration* of *applications*, *resources* and *devices*.

Those function block types shown in Annex A which utilize *execution control charts* are *basic function block types*. Where textual declarations of *algorithms* are given for these function block types, the language used is the Structured Text (ST) language defined in IEC 61131-3.

Reference implementations for some of the function block types in Annex A are given as *composite function block type* definitions. These implementations are normative only in the sense that the functional behaviors of compliant implementations shall be equivalent to those of the reference implementation, where the following considerations apply to the timing parameters defined in 4.5.3.

- The parameters T_{setup}, T_{start} and T_{finish} are considered to be zero (0) for all *component function blocks* in the reference implementation.
- The parameter T_{alg} is considered to be equal to the parameter `DT` for all instances of `E_DELAY` type used as *component function blocks* in the reference implementation, and to be zero (0) for all other component function blocks in the reference implementation.

All other function block types given in Annex A are *service interface function block types*.

NOTE Full textual specifications of all function block types shown in Table A.1 are given in Annex F.

Table A.1 – Event function blocks (1 of 6)

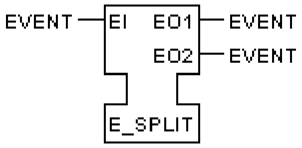
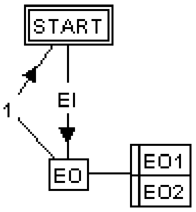
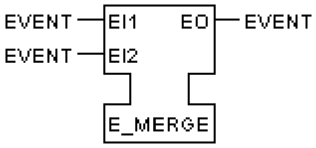
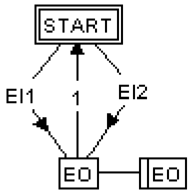
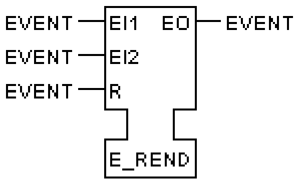
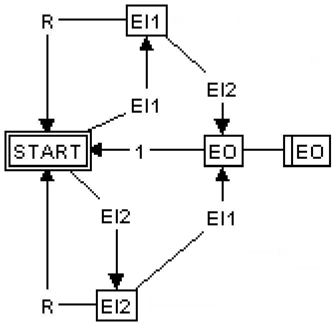
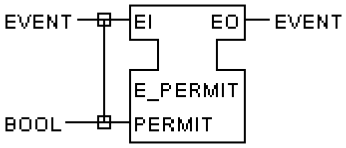
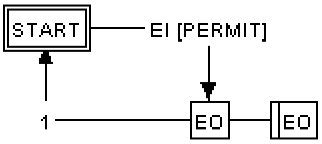
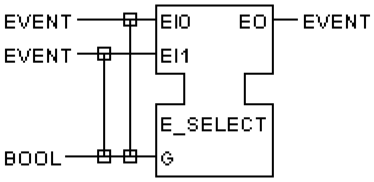
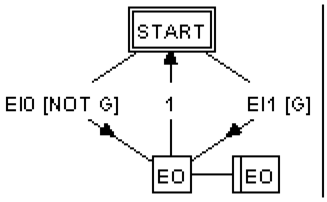
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>ECC/Algorithms/Service sequences</th></tr>
<tr><td>1</td><td colspan="2">Split an event</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
<tr><td colspan="3">The occurrence of an event at <code>EI</code> causes the occurrence of events at <code>EO1, EO2,...,EOn</code> (<code>n=2</code> in the above example).</td></tr>
<tr><td>2</td><td colspan="2">Merge (OR) of multiple events</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
<tr><td colspan="3">The occurrence of an event at any of the inputs <code>EI1, EI2,...,EIn</code> causes the occurrence of an event at <code>EO</code> (<code>n=2</code> in the above example).</td></tr>
<tr><td>3</td><td colspan="2">Rendezvous of two events</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
<tr><td>4</td><td colspan="2">Permissive propagation of an event</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
<tr><td>5</td><td colspan="2">Selection between two events</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
</table>

Table A.1 (2 of 6)

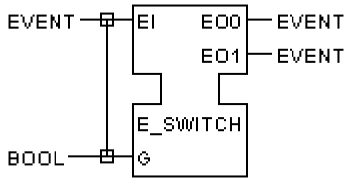
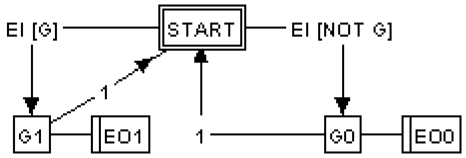
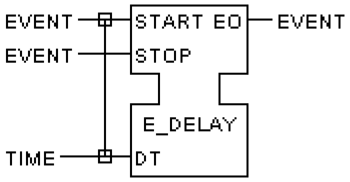
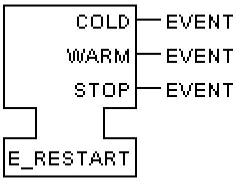
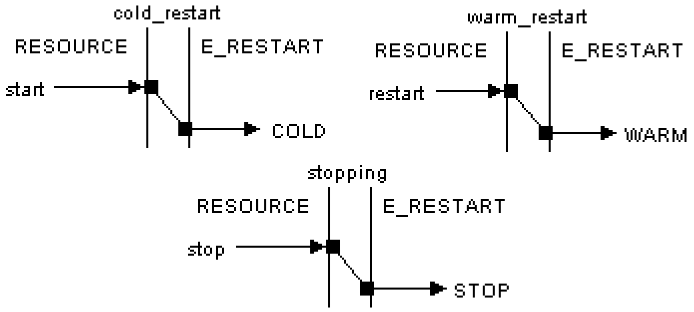
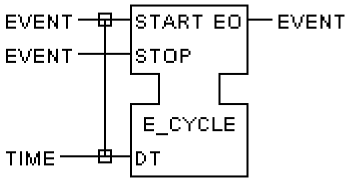
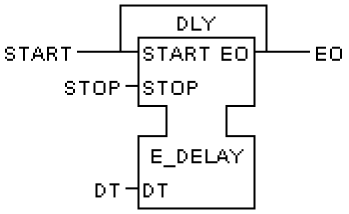
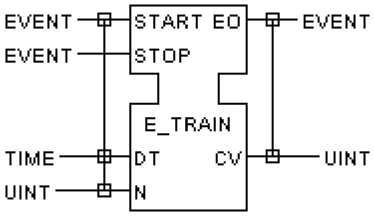
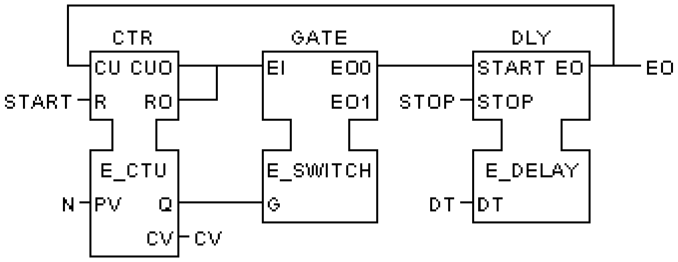
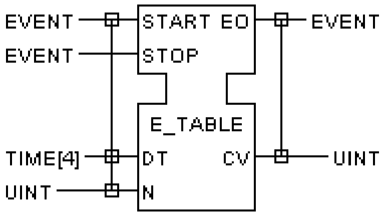
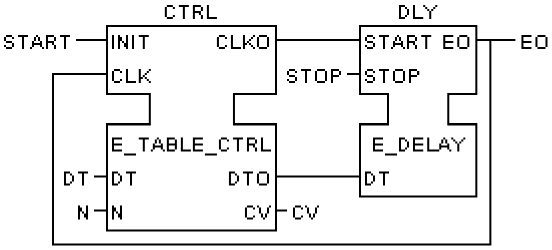
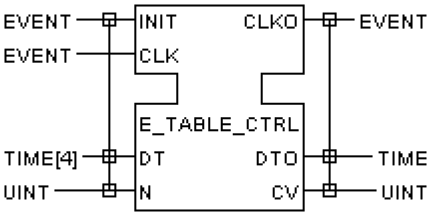
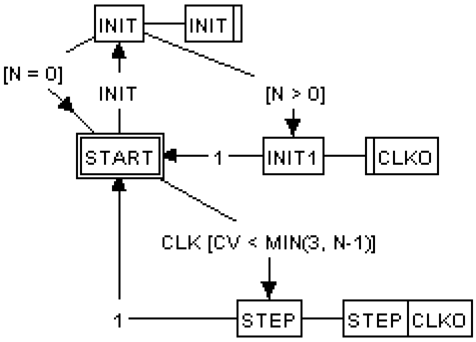
No.	Description	
	Interface	**ECC/Algorithms/Service sequences**
6	**Switching (demultiplexing) an event**	
		
7	**Delayed propagation of an event**	
		An event at `EO` is generated at a time interval `DT` after the occurrence of an event at the `START` input. The event delay is cancelled by an occurrence of an event at the `STOP` input. If multiple events occur at the `START` input before the occurrence of an event at `EO`, only a single event occurs at `EO`, at a time `DT` after the first event occurrence at the `START` input. No event delay will be initiated if an event occurs at the `START` input with a value of `DT` which is not greater than `t#0s`.
8	**Generation of restart events**	
		
	a) An event is issued at the COLD output upon "cold restart" of the associated resource. b) An event is issued at the WARM output upon "warm restart" of the associated resource. c) An event is issued at the STOP output (if possible) prior to "stopping" of the associated resource. NOTE 1 See IEC 61131-1 for a discussion of "cold restart" and "warm restart".	
9	**Periodic (cyclic) generation of an event**	
	 An event occurs at `EO` at an interval `DT` after the occurrence of an event at `START`, and at intervals of `DT` thereafter until the occurrence of an event at `STOP`.	

Table A.1 (3 of 6)

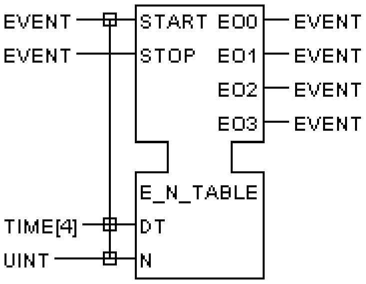
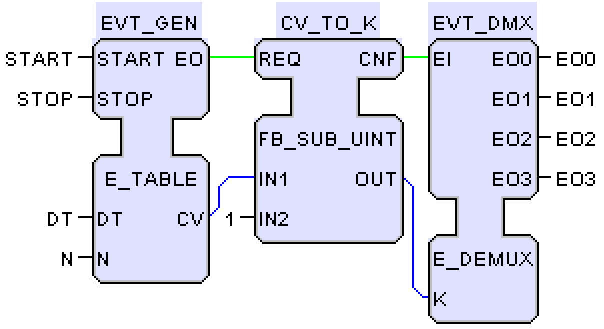
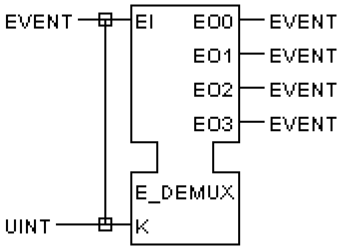
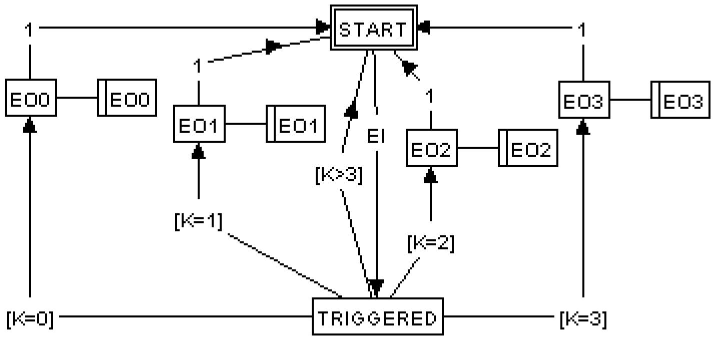
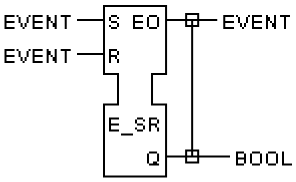
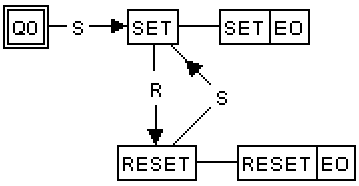
<table>
<tr><th>No.</th><th colspan="2">Description</th></tr>
<tr><th colspan="2">Interface</th><th>ECC/Algorithms/Service sequences</th></tr>
<tr><td>**10**</td><td colspan="2">**Generation of a finite train of events**</td></tr>
<tr><td colspan="2">
</td><td>

NOTE 2 See table entry 18 for a definition of the <code>E_CTU</code> type.</td></tr>
<tr><td colspan="3">An event occurs at <code>EO</code> at an interval <code>DT</code> after the occurrence of an event at <code>START</code>, and at intervals of <code>DT</code> thereafter, until <code>N</code> occurrences have been generated or an event occurs at the <code>STOP</code> input.
NOTE 3 The count <code>CV</code> is reset whenever an event occurs at the <code>START</code> interface, but the delay does not restart unless it is already stopped. This behavior maintains the inter-<code>EO</code> interval when restarting the count.</td></tr>
<tr><td>**11**</td><td colspan="2">**Generation of a finite train of events (table driven)**</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
<tr><td colspan="3">An event occurs at <code>EO</code> at an interval <code>DT[0]</code> after the occurrence of an event at <code>START</code>. A second event occurs at an interval <code>DT[1]</code> after the first, etc., until <code>N</code> occurrences have been generated or an event occurs at the <code>STOP</code> input. The current event count is maintained at the <code>CV</code> output.
NOTE 4 In this example implementation, <code>N <= 4</code>.
NOTE 5 Implementation using the <code>E_TABLE_CTRL</code> function block type illustrated below is not a normative requirement. Equivalent functionality can be implemented by various means.</td></tr>
<tr><td colspan="2">
</td><td>
</td></tr>
</table>

```
ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM
```

```
ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM
```

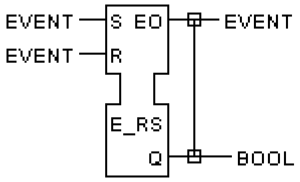
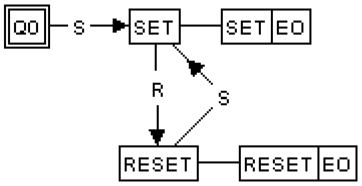
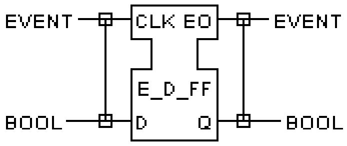
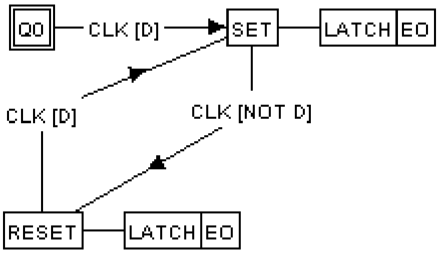
Table A.1 (4 of 6)

No.	**Description**	
	Interface	**ECC/Algorithms/Service sequences**
12	**Generation of a finite train of separate events (table driven)**	
		
	An event occurs at `EO0` at an interval `DT[0]` after the occurrence of an event at `START`. An event occurs at `EO1` an interval `DT[1]` after the occurrence of the event at `EO0`, etc., until `N` occurrences have been generated or an event occurs at the `STOP` input. NOTE 6 In this example implementation, `N <= 4`. NOTE 7 Implementation using the `E_DEMUX` function block type illustrated below is not a normative requirement. Equivalent functionality can be implemented by various means.	
		
13	**Event-driven bistable**	
		
	The output `Q` is set to `1 (TRUE)` upon the occurrence of an event at the `S` input, and is reset to `0 (FALSE)` upon the occurrence of an event at the `R` input. An event is issued at the `EO` output when the value of `Q` changes.	

```
ALGORITHM SET IN ST:   (* Set Q *)
  Q:= TRUE;
END_ALGORITHM
```

```
ALGORITHM RESET IN ST:  (* Reset Q *)
  Q:= FALSE;
END_ALGORITHM
```

Table A.1 (5 of 6)

No.	Description
Interface	**ECC/Algorithms/Service sequences**
14	**Event-driven bistable**
	
The output Q is set to 1 (TRUE) upon the occurrence of an event at the S input, and is reset to 0 (FALSE) upon the occurrence of an event at the R input. An event is issued at the EO output when the value of Q changes. NOTE 8 The implementation of this function block type is identical to E_SR. Both E_SR and E_RS are implemented for consistency with the SR and RS types of IEC 61131-3, although there is no "dominance" of events as there would be for level-controlled R and S inputs.	
15	**D (Data latch) bistable**
	

	`ALGORITHM LATCH IN ST:`
	`  Q:= D;`
	`END_ALGORITHM`

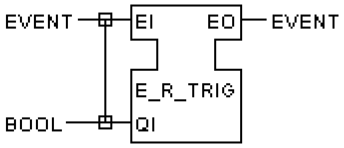
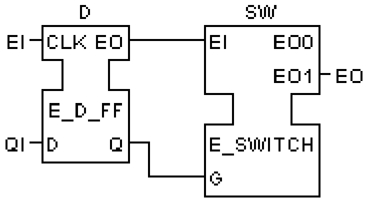
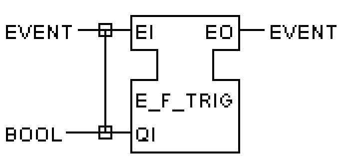
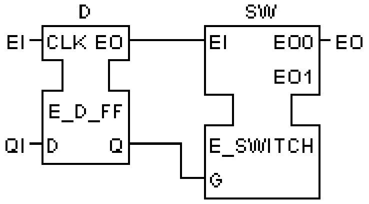
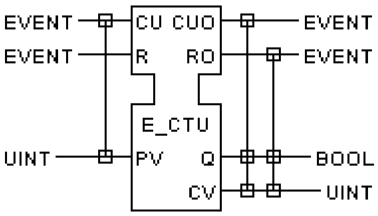
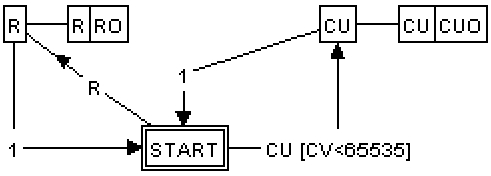
16	**Boolean rising edge detection**
	
17	**Boolean falling edge detection**
	

Table A.1 (6 of 6)

No.	Description	
Interface		**ECC/Algorithms/Service sequences**
18	**Event-driven up counter**	
		

```
ALGORITHM R IN ST: (* Reset *)
  CV:= 0;
  Q:= 0;
END_ALGORITHM
```

```
ALGORITHM CU IN ST: (* Count Up *)
  CV:= CV + 1;
  Q:= (CV >= PV);
END_ALGORITHM
```

Graphical shorthand notations may be substituted for the `E_SPLIT` and `E_MERGE` blocks defined in Table A.1. For example, the shorthand (implicit) representation shown in Figure A.1b is equivalent to the explicit representation in Figure A.1a.

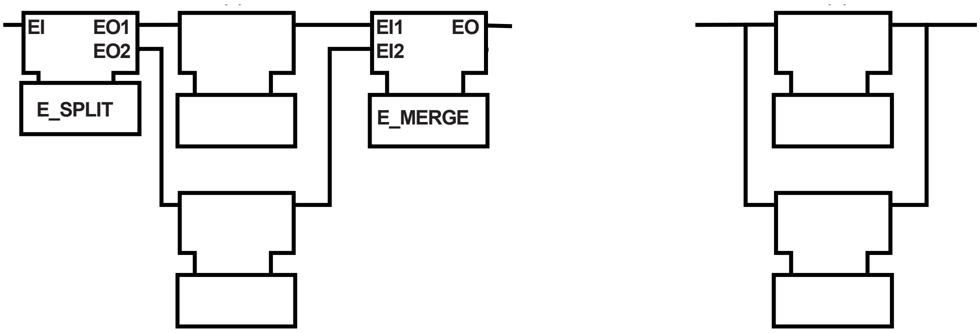


Figure A.1a – Explicit representation

Figure A.1b – Implicit representation

NOTE Irrelevant details are suppressed in the above figure.

Figure A.1 – Event split and merge

Annex B
(normative)

Textual syntax

B.1 Syntax specification technique

The textual constructs in Annex B are specified in terms of a *syntax*, which specifies the allowable combinations of symbols which can be used to define a program; and a set of *semantics*, which specify the meanings of the symbol combinations defined by the syntax.

A **syntax** is defined by a set of *terminal symbols* to be utilized for program specification; a set of *non-terminal symbols* defined in terms of the terminal symbols; and a set of *production rules* specifying those definitions.

The **terminal symbols** for textual specifications of entities defined in this standard consist of combinations of the characters in the character set given as Table 2 – Row 00 of the "Basic Latin to CJK Compatibility" table linked to Clause 33 defined in ISO/IEC 10646:2003.

For the purposes of this standard, terminal textual symbols consist of the appropriate character string enclosed in paired single or double quotes. For example, a terminal symbol represented by the character string ABC can be represented by either `"ABC"` or `'ABC'`.

This allows the representation of strings containing either single or double quotes; for instance, a terminal symbol consisting of the double quote itself would be represented by `'"'`

A special terminal symbol utilized in this syntax is the "null string", that is, a string containing no characters. This is represented by the terminal symbol `NIL`.

Non-terminal textual symbols are represented by strings of lower-case letters, numbers, and the underline character (_), beginning with a lower-case letter. For instance, the strings `nonterm1` and `non_term_2` are valid nonterminal symbols, while the strings `3nonterm` and `_nonterm4` are not.

The **production rules** given in this standard form an *extended grammar* in which each rule has the form

```
non_terminal_symbol::= extended_structure
```

This rule can be read as:

"A `non_terminal_symbol` can consist of an `extended_structure`."

Extended structures can be constructed according to the following rules:

a) The null string, `NIL`, is an extended structure.

b) A terminal symbol is an extended structure.

c) A non-terminal symbol is an extended structure.

d) If `S` is an extended structure, then the following expressions are also extended structures:

- `(S)`, meaning `S` itself.
- `{S}`, *closure*, meaning zero or more concatenations of `S`.

- `[S]`, *option*, meaning zero or one occurrence of `S`.

e) If `S1` and `S2` are extended structures, then the following expressions are extended structures:

- `S1 | S2`, *alternation*, meaning a choice of `S1` or `S2`.
- `S1 S2`, *concatenation*, meaning `S1` followed by `S2`.

f) Concatenation *precedes* alternation, that is, `S1 | S2 S3` is equivalent to `S1 | (S2 S3)`, and `S1 S2 | S3` is equivalent to `(S1 S2) | S3`.

Semantics are defined in this standard by appropriate natural language text, accompanying the production rules, which references the descriptions provided in the appropriate clauses. Standard options available to the user and vendor are specified in these semantics.

In some cases it is more convenient to embed semantic information in an extended structure. In such cases, this information is delimited by paired angle brackets, for example, `<semantic information>`.

B.2 Function block and subapplication type specification

B.2.1 Function block type specification

The syntax defined in B.2.1 can be used for the textual specification of *function block types* according to the rules given in Clauses 5 and 6 of this standard.

SYNTAX:

```
fb_type_declaration::=
     'FUNCTION_BLOCK' fb_type_name
     fb_interface_list
     [fb_internal_variable_list] <only for basic FB>
     [fb_instance_list]      <only for composite FB>
     [plug_list]
     [socket_list]
     [fb_connection_list]      <only for composite FB>
     [fb_ecc_declaration]      <only for basic FB>
     {fb_algorithm_declaration}  <only for basic FB>
     [fb_service_declaration]
     'END_FUNCTION_BLOCK'


fb_interface_list::=
     [event_input_list]
     [event_output_list]
     [input_variable_list]
     [output_variable_list]


event_input_list::=
     'EVENT_INPUT'
     {event_input_declaration}
     'END_EVENT'


event_output_list::=
     'EVENT_OUTPUT'
     {event_output_declaration}
     'END_EVENT'


event_input_declaration::= event_input_name [ ':' event_type ]
     ['WITH' input_variable_name {',' input_variable_name}] ';'


event_output_declaration::= event_output_name [ ':' event_type ]
     ['WITH' output_variable_name {',' output_variable_name}] ';'
```

```
input_variable_list::=
     'VAR_INPUT' {input_var_declaration ';'} 'END_VAR'

output_variable_list::=
     'VAR_OUTPUT' {output_var_declaration ';'} 'END_VAR'

fb_internal_variable_list::=
     'VAR' {internal_var_declaration ';'} 'END_VAR'

input_var_declaration::=
     input_variable_name {',' input_variable_name} ':' var_spec_init

output_var_declaration::=
     output_variable_name {',' output_variable_name} ':' var_spec_init

internal_var_declaration::=
     internal_variable_name {',' internal_variable_name}
     ':' var_spec_init

var_spec_init::= located_var_spec_init <as specified in IEC 61131-3>

fb_instance_list::= 'FBS'
     {fb_instance_definition ';'}
     'END_FBS'

fb_instance_definition::= fb_instance_name ':' fb_type_name [parameters]

plug_list::= 'PLUGS'
     {plug_name ':' adapter_type_name [parameters] ';'}
     'END_PLUGS'

socket_list::= 'SOCKETS'
     {socket_name ':' adapter_type_name [parameters] ';'}
     'END_SOCKETS'

fb_connection_list::=   <may be empty, e.g. for basic FB>
     [event_conn_list]
     [data_conn_list]
     [adapter_conn_list]

event_conn_list::=
     'EVENT_CONNECTIONS'
     {event_conn}
     'END_CONNECTIONS'

event_conn::= event_conn_source 'TO' event_conn_destination ';'


event_conn_source::= ([plug_name '.'] event_input_name)
     | ((fb_instance_name | socket_name) '.' event_output_name)

event_conn_destination::= ([plug_name '.'] event_output_name)
     | ((fb_instance_name | socket_name) '.' event_input_name)

data_conn_list::=
     'DATA_CONNECTIONS'
     {data_conn}
     'END_CONNECTIONS'

data_conn::= data_conn_source 'TO' data_conn_destination ';'
```

```
data_conn_source::= ([plug_name '.'] input_variable_name)
     | ((fb_instance_name | socket_name) '.' output_variable_name)

data_conn_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_instance_name | socket_name) '.' input_variable_name)

adapter_conn_list::=
     'ADAPTER_CONNECTIONS'
     {adapter_conn}
     'END_CONNECTIONS'

adapter_conn::=
     ((fb_instance_name '.' plug_name ) | socket_name)
     'TO' ((fb_instance_name '.' socket_name ) | plug_name) ';'

fb_ecc_declaration::=
     'EC_STATES'
     {ec_state}    <first state is initial state>
     'END_STATES'
     'EC_TRANSITIONS'
     {ec_transition}
     'END_TRANSITIONS'

ec_state::= ec_state_name
     [':' ec_action {',' ec_action}] ';'

ec_action::= algorithm_name | ('->' ec_action_output)
     | (algorithm_name '->' ec_action_output)


ec_action_output:= ([plug_name '.'] event_output_name)
     | (socket_name '.' event_input_name)

ec_transition::=
     ec_state_name
     'TO' ec_state_name
     ':=' ec_transition_condition ';'

ec_transition_condition::= '1'
     | ec_transition_event | '[' guard_condition ']'
     | ec_transition_event '[' guard_condition ']'


ec_transition_event::= ([plug_name '.'] event_input_name)
     | (socket_name '.' event_output_name)

guard_condition::= expression <over ec_expression_operand elements>
     <as defined in IEC 61131-3>
     <Shall evaluate to a BOOL value>

ec_expression_operand::=
     ([(plug_name | socket_name) '.'] input_variable_name)
     | ([(plug_name | socket_name) '.'] output_variable_name)
     | internal_variable_name
     | constant

fb_algorithm_declaration::=
     'ALGORITHM' algorithm_name 'IN' language_type ':'
     [temp_var_decls]
     algorithm_body
     'END_ALGORITHM'

temp_var_decls::= <as defined in IEC 61131-3>
```

```
algorithm_body::= <as defined in compliant standards>

fb_service_declaration::=
     'SERVICE' service_interface_name '/' service_interface_name
     {service_sequence}
     'END_SERVICE'

service_interface_name::= fb_type_name | 'RESOURCE'

service_sequence::=
     'SEQUENCE' sequence_name
     {service_transaction ';'}
     'END_SEQUENCE'

service_transaction::=
     [input_service_primitive] '->' output_service_primitive
     {'->' output_service_primitive}

input_service_primitive::= service_interface_name '.'
     ([plug_name '.'] event_input_name
      | socket_name '.' event_output_name)
     ['+' | '-']
     '(' [input_variable_name {',' input_variable_name}] ')'

output_service_primitive::= service_interface_name '.' ('NULL' |
     ([plug_name '.'] event_output_name
      | socket_name '.' event_input_name)
     ['+' | '-']
     '(' [output_variable_name {',' output_variable_name}] ')')

algorithm_name::= identifier

ec_state_name::= identifier

event_input_name::= identifier

event_output_name::= identifier

event_type::= identifier

fb_instance_name::= identifier

fb_type_name::= identifier

input_variable_name::= identifier

internal_variable_name::= identifier

language_type::= identifier

output_variable_name::= identifier

plug_name::= identifier

sequence_name::= identifier

socket_name::= identifier
```

B.2.2 Subapplication type specification

The syntax defined in this subclause can be used for the textual specification of *subapplication types* according to the rules given in 5.4.1.

The productions given in B.2.1 also apply to this subclause.

SYNTAX:

```
subapplication_type_declaration::=
     'SUBAPPLICATION' subapp_type_name
          subapp_interface_list
          [fb_instance_list]
          [subapp_instance_list]
          [plug_list]
          [socket_list]
          [subapp_connection_list]
     'END_SUBAPPLICATION'

subapp_interface_list::=
     [subapp_event_input_list]
     [subapp_event_output_list]
     [input_variable_list]
     [output_variable_list]

subapp_event_input_list::=
     'EVENT_INPUT'
     {subapp_event_input_declaration}
     'END_EVENT'

subapp_event_output_list::=
     'EVENT_OUTPUT'
     {subapp_event_output_declaration}
     'END_EVENT'

subapp_event_input_declaration::=
     event_input_name [ ':' event_type ] ';'

subapp_event_output_declaration::=
     event_output_name [ ':' event_type ] ';'

subapp_instance_list::= 'SUBAPPS'
     {subapp_instance_definition ';'}
     'END_SUBAPPS'

subapp_instance_definition::= subapp_instance_name ':' subapp_type_name

subapp_connection_list::=
     [subapp_event_conn_list]
     [subapp_data_conn_list]
     [adapter_conn_list]

subapp_event_conn_list::=
     'EVENT_CONNECTIONS'
     {subapp_event_conn}
     'END_CONNECTIONS'

subapp_event_conn::= subapp_event_source 'TO' subapp_event_destination ';'

subapp_event_source:: = ([plug_name '.'] event_input_name)
     | ((fb_subapp_name | socket_name) '.' event_output_name
```

```
subapp_event_destination::= ([plug_name '.'] event_output_name)
     | ((fb_subapp_name | socket_name) '.' event_input_name)

fb_subapp_name::= fb_instance_name | subapp_instance_name

subapp_data_conn_list::=
     'DATA_CONNECTIONS'
     {subapp_data_conn}
     'END_CONNECTIONS'

subapp_data_conn::= subapp_data_source 'TO' subapp_data_destination ';'

subapp_data_source::= ([plug_name '.'] input_variable_name)
     | ((fb_subapp_name | socket_name) '.' output_variable_name)

subapp_data_destination::= ([plug_name '.'] output_variable_name)
     | ((fb_subapp_name | socket_name) '.' input_variable_name)

subapp_type_name::= identifier

subapp_instance_name::= identifier
```

B.3 Configuration elements

The syntax defined in this clause can be used for the textual specification of *resource types, device types, segment types, applications,* and *system configurations* according to the rules given in Clause 7.

The productions given in Clause B.2 also apply to this clause.

SYNTAX:

```
application_configuration::=
     'APPLICATION' application_name
     [fb_instance_list]
     [subapp_instance_list]
     [subapp_connection_list]
     'END_APPLICATION'

system_configuration::= 'SYSTEM' system_name
      {application_configuration}
      device_configuration
      {device_configuration}
      [mappings]
      [segments]
      [links]
     'END_SYSTEM'

segments::= 'SEGMENTS'
      segment
      {segment}
     'END_SEGMENTS'

segment::= segment_name ':' segment_type_name [parameters] ';'

links::= 'LINKS'
      link
      {link}
     'END_LINKS'
```

```
link::= resource_hierarchy '=>' segment_name [parameters] ';'

parameters::= '(' parameter {',' parameter} ')'

parameter::= parameter_name ':='
     (constant | enumerated_value | array_initialization |
     structure_initialization) ';'
     <as defined in IEC 61131-3>

device_configuration::=
     'DEVICE' device_name ':' device_type_name [parameters]
     [resource_type_list]
     {resource_configuration}
     [fb_instance_list]
     [config_connection_list]
     'END_DEVICE'

resource_type_list::= 'RESOURCE_TYPES'
     {resource_type_name ';'}
     'END_RESOURCE_TYPES'

resource_configuration::=
     'RESOURCE' resource_instance_name ':' resource_type_name [parameters]
     [fb_type_list]
     [fb_instance_list]
     [config_connection_list]
     'END_RESOURCE'

fb_type_list::= 'FB_TYPES' {fb_type_name ';'} 'END_FB_TYPES'

config_connection_list::=
     [config_event_conn_list]
     [config_data_conn_list]
     [config_adapter_conn_list]

config_event_conn_list::= 'EVENT_CONNECTIONS'
     {config_event_conn}
     'END_CONNECTIONS'

config_event_conn::= fb_instance_name '.' event_output_name
     'TO' fb_instance_name '.' event_input_name ';'

config_data_conn_list::= 'DATA_CONNECTIONS'
     {config_data_conn}
     'END_CONNECTIONS'

config_data_conn::=
     (fb_instance_name '.' output_variable_name | input_variable_name)
     'TO'
     (fb_instance_name | resource_instance_name)'.' input_variable_name ';'
     <resource_instance_name only applies to connections within device_type or
     device_configuration declarations>

config_adapter_conn_list::= 'ADAPTER_CONNECTIONS'
     {config_adapter_conn}
     'END_CONNECTIONS'

config_adapter_conn::= fb_instance_name '.' plug_name
     'TO' fb_instance_name '.' socket_name ';'

fb_instance_reference::= [app_hierarchy_name] fb_instance_name

app_hierarchy_name:= application_name '.'{subapp_instance_name '.'}
```

```
device_type_specification::=
     'DEVICE_TYPE' device_type_name
     [input_variable_list]
     [resource_type_list] <if not given, defined by resource instances>
     {resource_instance}
     [fb_instance_list]
     [config_connection_list]
     'END_DEVICE_TYPE'

resource_instance::=
     'RESOURCE' resource_instance_name ':' resource_type_name
     [fb_instance_list]
     [config_connection_list]
     'END_RESOURCE'

resource_type_specification::= 'RESOURCE_TYPE' resource_type_name
     [input_variable_list]
     [fb_type_list] <if not given, defined by function block instances>
     [fb_instance_list]
     config_connection_list
     'END_RESOURCE_TYPE'

segment_type_specification::= 'SEGMENT_TYPE' segment_type_name
     {parameter_declaration}
     'END_SEGMENT_TYPE'

parameter_declaration:= parameter_name ':' var_spec_init ';'

mappings::= 'MAPPINGS' mapping {mapping} 'END_MAPPINGS'

mapping::= fb_instance_reference 'ON' fb_resource_reference ';'

fb_resource_reference::= resource_hierarchy ['.' fb_instance_name]
     <When the optional element ['.' fb_instance_name] is not given, the
     instance name of the FB in the resource is the same as its instance name
     in the corresponding fb_instance_reference of the mapping.>

resource_hierarchy::= device_name ['.' resource_instance_name]

segment_name::= identifier

segment_type_name::= identifier

parameter_name::= identifier

system_name::= identifier

device_name::= identifier

device_type_name::= identifier

application_name::= identifier

resource_instance_name::= identifier

resource_type_name::= identifier
```

B.4 Common elements

Where syntactic productions are not given for non-terminal symbols in Annex B, the syntactic productions and corresponding semantics given in Annex B of IEC 61131-3:2003 shall apply.

B.5 Supporting productions for management commands

The syntax defined in this clause is referenced in Table 8.

SYNTAX:

```
data_type_list::= 'DATA_TYPES' {data_type_name ';'} 'END_DATA_TYPES'

connection_definition::=
     connection_start_point ' ' connection_end_point

connection_start_point::= fb_instance_reference '.' attachment_point

connection_end_points::=
     connection_end_point {',' connection_end_point}

connection_end_point::= fb_instance_reference '.' attachment_point

attachment_point::= identifier

referenced_parameter::=
     [(resource_instance_name | fb_instance_name)'.'] parameter
     <resource_instance_name refers to a resource located in the same device
     as the MANAGER block defined in 6.3.2>
     <fb_instance_name refers to an FB contained in the same device or
     resource as the <MANAGER> block>
     <if no resource or FB instance name is given, the parameter refers to a
     parameter of the device or resource containing the MANAGER block>

parameter_reference::=
     [(resource_instance_name | fb_instance_name)'.'] parameter_name
     <see above for semantics>

all_data_types::= 'ALL_DATA_TYPES'

all_fb_types::= 'ALL_FB_TYPES'

fb_status::= 'IDLE' | 'RUNNING' | 'STOPPED' | 'KILLED'
```

B.6 Tagged data types

The syntax defined below shall be used for the assignment of tags as defined in ISO/IEC 8824-1 to derived data types defined as specified in Annex B and Annex E. As defined in ISO/IEC 8824-1, the class tags `APPLICATION` and `PRIVATE` shall be used except for types to be used only in context-specific tagging.

SYNTAX:

```
tagged_type_declaration::=
     'TYPE'
     asn1_tag type_declaration ';'
     {asn1_tag type_declaration ';'}
     'END_TYPE'

asn1_tag::= '[' ['APPLICATION' | 'PRIVATE'] (integer | hex_integer) ']'
```

B.7 Adapter interface types

See 5.5 for the semantics associated with the following syntax.

SYNTAX:

```
adapter_type_declaration::=
     'ADAPTER' adapter_type_name
          fb_interface_list
          [fb_service_declaration]
     'END_ADAPTER'

adapter_type_name::= identifier
```

Annex C
(informative)

Object models

C.1 Model notation

Annex C presents object models for some of the classes which may be used in Engineering Support Systems (ESS) to support the design, implementation, commissioning and operation of Industrial-Process Measurement and Control Systems (IPMCSs) constructed according to the architecture defined in this standard.

The notation used in Annex C is the Unified Modeling Language (UML). References to extensive documentation of this notation can be found on the Internet at the Uniform Resource Locator (URL) **http://www.omg.org/uml/**.

C.2 ESS models

C.2.1 ESS overview

Figure C.1 presents an overview of the major classes in the ESS (Engineering Support System) for an industrial-process measurement and control system (IPMCS), and their correspondence to the classes of objects in the IPMCS. Descriptions of the classes in Figure C.1 are given in Table C.1.

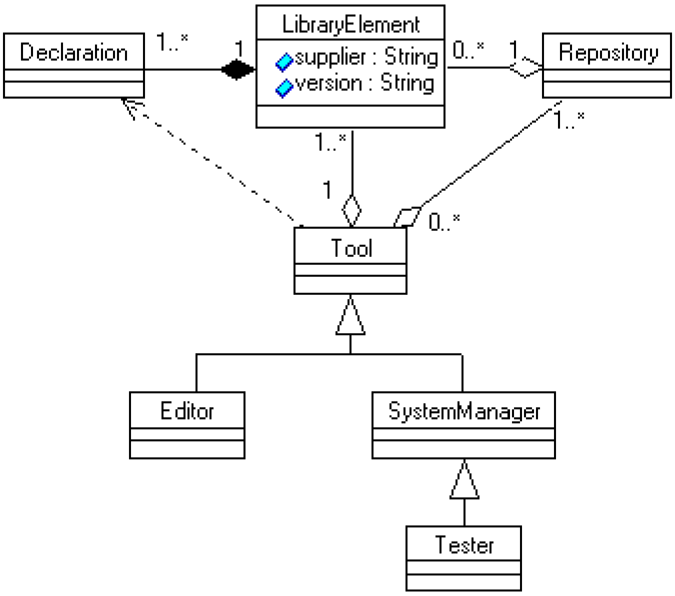


Figure C.1 – ESS overview

Table C.1 – ESS class descriptions

Declaration	This is the abstract superclass for *declarations*.
Editor	Instances of this class provide the editing functions on *declarations* necessary to support the EDIT use case.
LibraryElement	This is the abstract superclass of objects which may be stored in repositories and which may be imported and exported in the textual syntax defined in Annex B, or the XML syntax defined in IEC 61499-2. Such objects have supplier (vendor, programmer, etc.) and version(version number, date, etc.) attributes to assist in management, in addition to a name (inherited from **NamedDeclaration –** see C.2.2) as a key attribute.
Repository	Instances of this class provide persistent storage and retrieval of library elements. They may also provide version control services.
SystemManager	Instances of this class provide the functions necessary to support the INSTALL and OPERATE use cases.
Tester	This class extends the capabilities of the SystemManager class to support the operations of the TEST use case.
Tool	This class models the generic behaviors of *software tools* for engineering support of IPMCSs.

C.2.2 Library elements

The subclasses of **LibraryElement** are shown in Figure C.2. The syntactic production in Annex B corresponding to each subclass is listed in Table C.2.

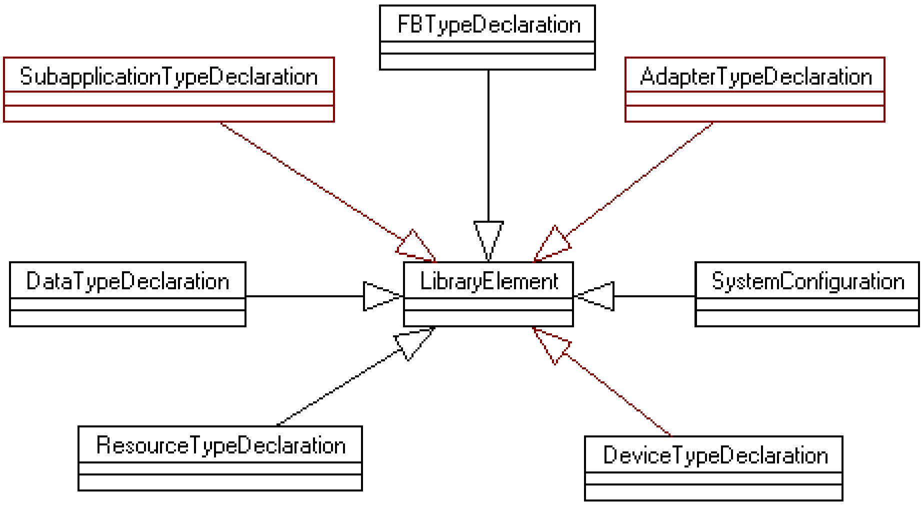


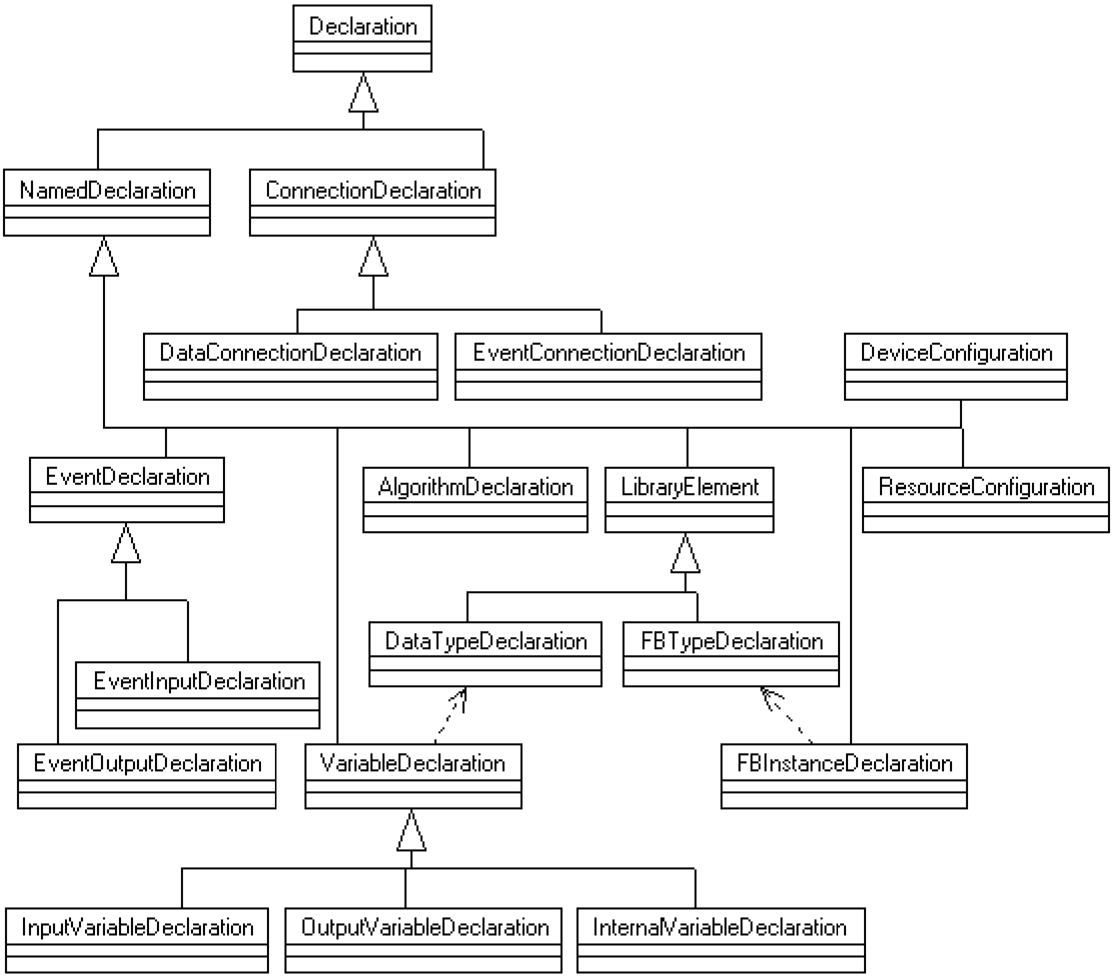
Figure C.2 – Library elements

Table C.2 – Syntactic productions for library elements

Class	Syntactic production
DataTypeDeclaration	type_declaration
FBTypeDeclaration	fb_type_declaration
AdapterTypeDeclaration	adapter_type_declaration
SubapplicationTypeDeclaration	subapplication_type_declaration
ResourceTypeDeclaration	resource_type_specification
DeviceTypeDeclaration	device_type_specification
SystemConfiguration	system_configuration

C.2.3 Declarations

Figure C.3 shows the class hierarchy of *declarations* which may be manipulated by *software tools*. The syntactic productions in Annex B corresponding to each of these subclasses are listed in Table C.3.



NOTE To avoid clutter, classes related to *adapter types, instances* and *connections* are not shown in this Figure; however, they are listed in Table C.3 for reference.

Figure C.3 – Declarations

Table C.3 – Syntactic productions for declarations

Class	Syntactic production
AdapterConnectionDeclaration	adapter_conn
AdapterTypeDeclaration	adapter_type_declaration
AlgorithmDeclaration	fb_algorithm_declaration
DataConnectionDeclaration	data_conn
DeviceConfiguration	device_configuration
EventConnectionDeclaration	event_conn
EventInputDeclaration	event_input_declaration
EventOutputDeclaration	event_output_declaration
FBInstanceDeclaration	fb_instance_definition
InputVariableDeclaration	input_var_declaration
InternalVariableDeclaration	internal_var_declaration
OutputVariableDeclaration	output_var_declaration
PlugDeclaration	Part of plug_list
ResourceConfiguration	resource_instance
SocketDeclaration	Part of socket_list

C.2.4 Function block network declarations

Figure C.4 shows the relationships among the elements of *function block network declarations*. See C.2.2 for definitions of the aggregated classes in this diagram.

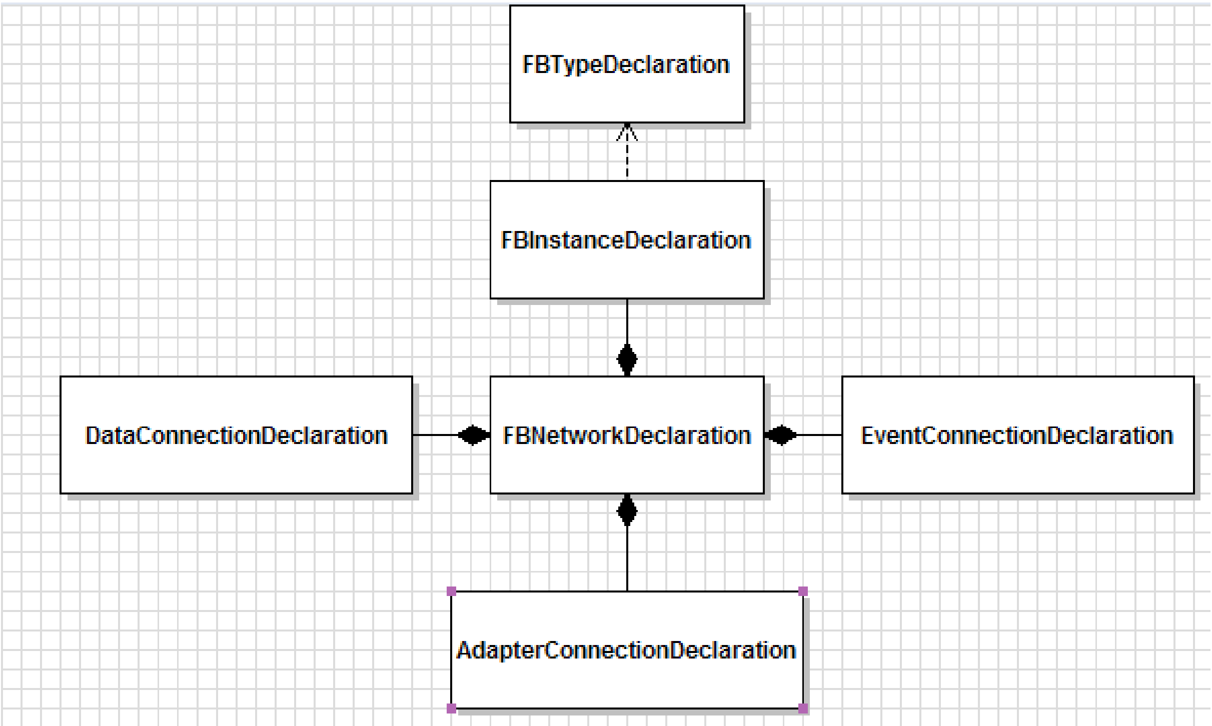


Figure C.4 – Function block network declarations

C.2.5 Function block type declarations

Figure C.5 shows the relationships among the elements of *function block type declarations*. Syntactic productions for the classes **EventInputDeclaration, EventOutputDeclaration, InputVariableDeclaration, OutputVariableDeclaration, InternalVariableDeclaration,** and the component classes of **FBNetworkDeclaration** are given in Table C.3. The syntactic productions `fb_ecc_declaration` and `fb_service_declaration` in Clause B.2 correspond to classes **ECCDeclaration** and **ServiceDeclaration**, respectively.

NOTE 1 Declarations of *subapplications* are represented by instances of the class **CompositeFBTypeDeclaration** which contain no event `WITH` data associations.

NOTE 2 **NamedDeclaration** is the abstract superclass of declarations which have names, e.g., *type names* or *instance names*.

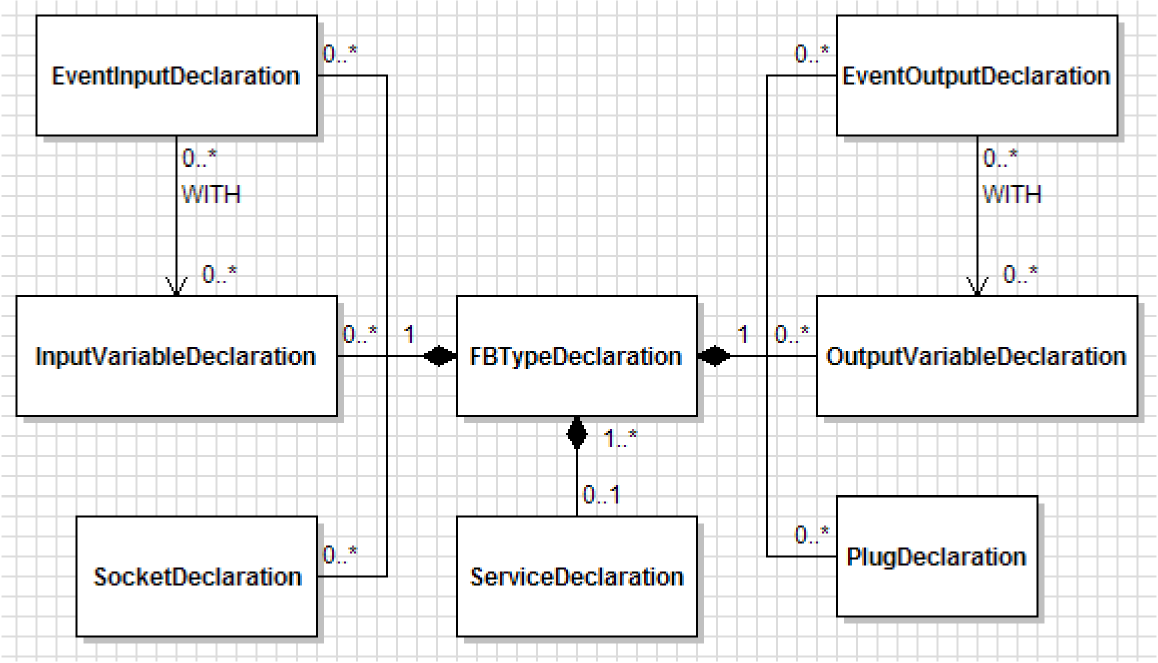


Figure C.5a – Composition

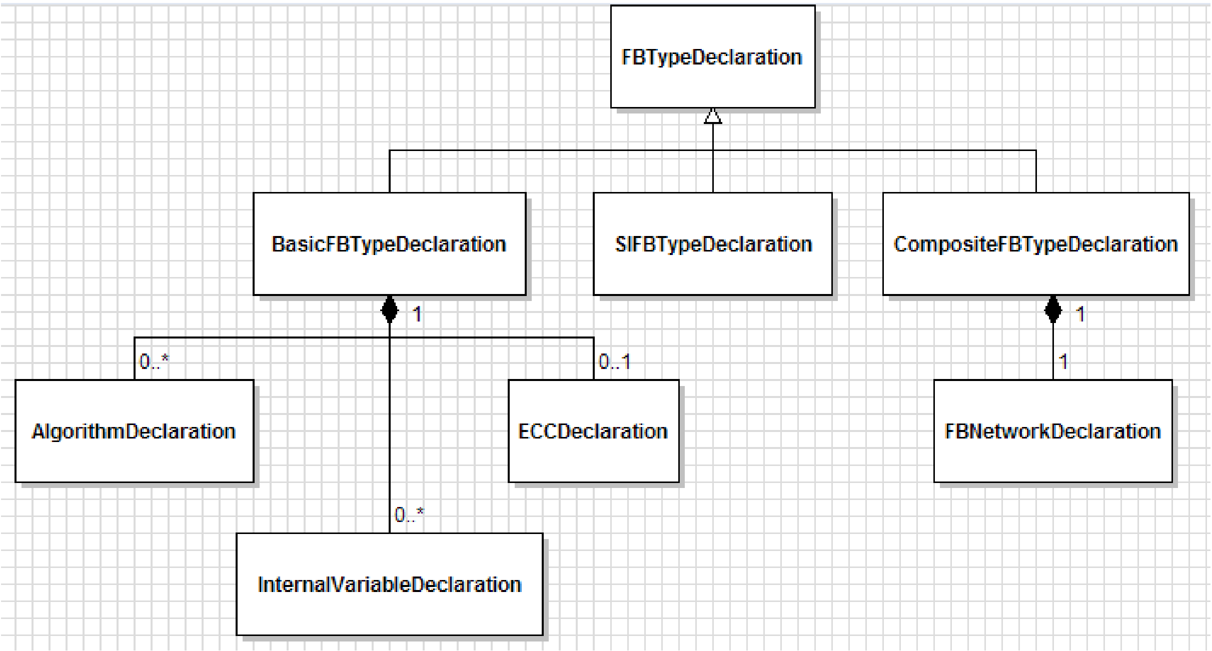


Figure C.5b – Class hierarchy

Figure C.5 – Function block type declarations

C.3 IPMCS models

Figure C.6 presents an overview of the major classes in the industrial-process measurement and control system (IPMCS). Descriptions of the classes in Figure C.6 and their corresponding objects in the Engineering Support System (ESS) are given in Table C.4.

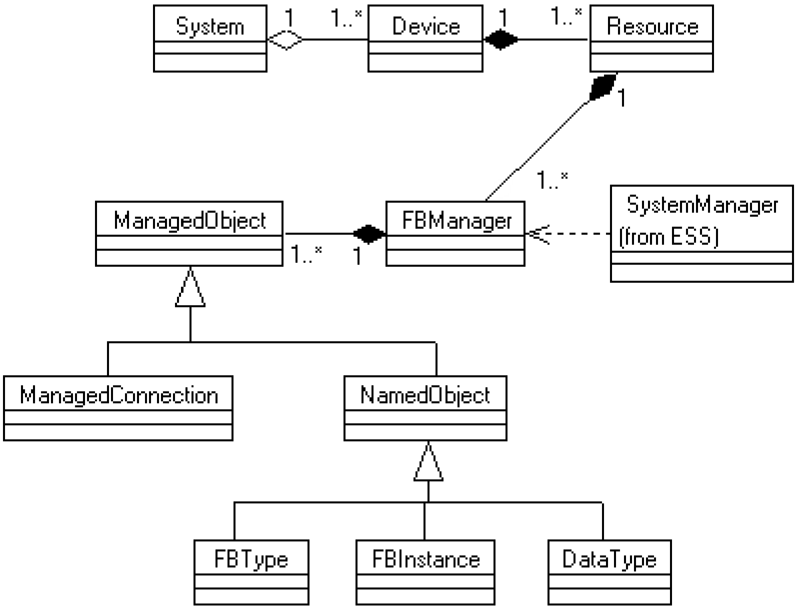


Figure C.6 – IPMCS overview

Table C.4 – IPMCS classes

IPMCS class	Description	Corresponding ESS class
DataType	An instance of this class is a *data type*.	**DataTypeDeclaration**
Device	An instance of this class represents a *device*.	**DeviceConfiguration**
FBInstance	An instance of this class is a *function block instance*.	**FBInstanceDeclaration**
FBManager	An instance of this class provides the management services defined in Clause 6.	**SystemManager**
FBType	An instance of this class is a *function block type*.	**FBTypeDeclaration**
ManagedConnection	Instances of this class can be accessed by an instance of the **FBManager** class using the source and destination combination as a unique key.	**ConnectionDeclaration**
ManagedObject	This is the abstract superclass of objects which are managed by an instance of the **FBManager** class. Such objects may have supplier (vendor, programmer, etc.) and version (version number, date, etc.) attributes to assist in management.	none
NamedObject	This is the abstract superclass of objects which can be accessed by name by an instance of the **FBManager** class.	**NamedDeclaration**
Resource	An instance of this class represents a *resource*.	**ResourceConfiguration**
System	An instance of this class represents an Industrial-Process Measurement and Control System (IPMCS).	**SystemConfiguration**

Figure C.7 shows the relationships among the elements of a *function block instance* and its associated *function block type*.

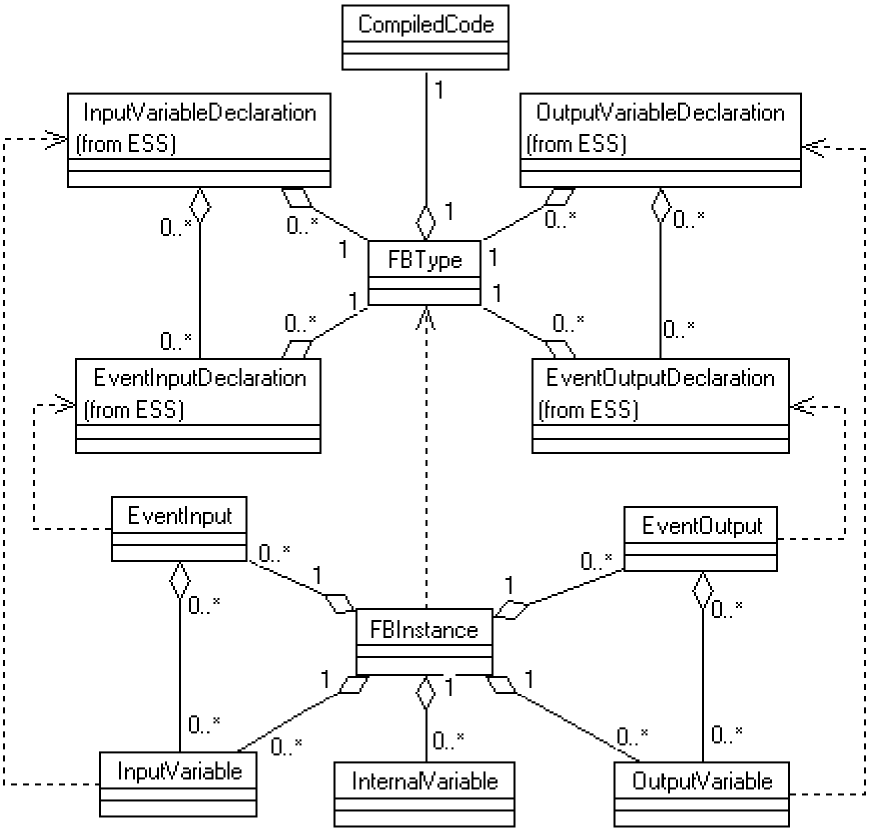


Figure C.7 – Function block types and instances

Annex D
(informative)

Relationship to IEC 61131-3

D.1 General

Functions and *function blocks* as defined in IEC 61131-3 can be used for the *declaration* of *algorithms* in *basic function block types* as specified in 5.2.1. Clause D.2 defines rules for the conversion of IEC 61131-3 *functions* and *function block types* into *simple function block types* so that they can be used in the specification of *applications* and *resource types.* Clause D.3 defines event-driven versions of IEC 61131-3 functions and function blocks for the same uses.

D.2 "Simple" function blocks

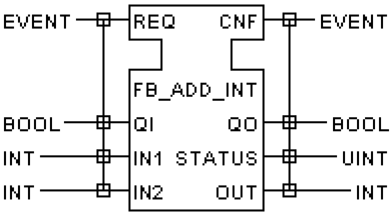
As illustrated in Figure D.1, IEC 61131-3 functions and function blocks can be converted to “simple” function blocks according to the following rules:

a) Simple function blocks are represented as *service interface function blocks* for application-initiated interactions as shown in Figure 21a.

b) The *type name* of the simple function block type is the name of the converted IEC 61131-3 function or function block type with the prefix `FB_` (for instance, `FB_ADD_INT` in Figure D.1). The prefix `F_` instead of `FB_` may optionally be used for simple function block types that are the result of conversions of IEC 61131-3 *functions*.

c) The *input* and *output variables* and their corresponding *data types* are the same as the corresponding input and output variables of the converted IEC 61131-3 function or function block type.

d) The `INIT` event input and `INITO` event output are used with simple function block types that have been converted from IEC 61131-3 *function block types*, and are not used with simple function block types that have been converted from IEC 61131-3 *functions*.



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

Figure D.1 – Example of a “simple” function block type

The behavior of *instances* of simple function block *types* is according to the following rules:

e) Initialization is as specified in 2.4.2 of IEC 61131-3:2003 for *variables*, and as specified in 2.6 of IEC 61131-3:2003 for Sequential Function Chart (SFC) elements.

f) The occurrence of an INIT+ service primitive is equivalent to "cold restart" initialization as defined in the above mentioned subclauses of IEC 61131-3:2003, followed by an INITO+ service primitive with a STATUS value of zero (0).

g) The occurrence of an INIT- or REQ- service primitive has no effect except to cause an INITO- or CNF-service primitive, respectively, with a STATUS value of one (1).

h) The occurrence of a REQ+ service primitive causes the *execution* of the *algorithm* specified in the function block body, according to the rules given in IEC 61131-3 for the language in which the algorithm is programmed.

i) Successful execution of the algorithm in response to a REQ+ primitive results in a CNF+ primitive with a STATUS value of zero (0).

j) If an error occurs during the execution of the algorithm, the result is a CNF- primitive with a STATUS value determined according to Table D.1.

Table D.1 – Semantics of STATUS values

Value	Semantics
0	Normal operation
1	`INIT-` or `REQ-` propagation
2	Type conversion error
3	Numerical result exceeds range for data type
4	Division by zero
5	Selector (`K`) out of range for `MUX` function
6	Invalid character position specified
7	Result exceeds maximum string length
8	Simultaneously true, non-prioritized transitions in a selection divergence
9	Action control contention error
10	Return from function without value assigned
11	Iteration fails to terminate
12	Invalid subscript value
13	Array size error

D.3 Event-driven functions and function blocks

IEC 61131-3 *functions* can be converted into function blocks for efficient use in event-driven systems according to the rules given in Clause D.2 with the following modifications:

a) the *type name* of the event-driven function block type is the same as the name of the converted IEC 61131-3 function with the additional prefix `E_`, e.g., `E_ADD_INT`;

b) a `CNF+` or `CNF-` primitive does not follow execution of the algorithm unless such execution results in a changed value of the function output.

NOTE If "daisy-chaining" of `CNF` outputs to `REQ` inputs is used to implement a sequence of calculations, then the sequence will stop at the first point where an output value does not change.

In general, since IEC 61131-3 *function blocks* have internal state information, such blocks shall be specially converted for use in event-driven systems. For instance, the `E_DELAY` function block shown in Table A.1 can be used for many of the delay functions provided by the timer function blocks in IEC 61131-3. An example of a conversion of the standard IEC 61131-3 `CTU` function block is given as Feature 18 of Table A.1.

D.4 Compliance with IEC 61131-3

Implementations of this standard shall comply with the requirements of the subclauses 1.5.1, 2.1, 2.2, 2.3 and 2.4 of IEC 61131-3:2003, and the associated elements of Annex B of IEC 61131-3:2003 for the syntax and semantics of textual representation of common elements, with the exceptions and extensions noted in Clause D.5.

Where syntactic productions are not given for non-terminal symbols in Annex B, the corresponding syntactic productions given in Annex B of IEC 61131-3:2003 shall apply.

D.5 Exceptions

Implementations of this standard shall **not** utilize the *directly represented variable* notation defined in 2.4.1.1 of IEC 61131-3:2003 and related features in other subclauses. However, a *literal* of `STRING` or `WSTRING` type, containing a string whose syntax and semantics correspond to the directly represented variable notation, may be used as a *parameter* of a *service interface function block* which provides access to the corresponding variable.

D.6 Interoperation with programmable controllers

D.6.1 Overview

A programmable controller may act as a *server,* as defined in IEC 61131-5, to a *device* as defined in this standard, acting as a *client* as defined in IEC 61131-5. These services are provided using the means defined in IEC 61131-5, and are accessed from the IEC 61499 device using instances of the *function block types* specified in Annex D. These function block types are modeled as *communication function block types* as defined in this standard.

The IEC 61499 client device may exist on a communication network along with the programmable controller acting as a server, or may be an implementer-specific subsystem within the “main processing unit” of the programmable controller, as illustrated in Figure 4 of IEC 61131-5:2000. In either case, the interaction between the IEC 61499 client device and the main processing unit is modelled as occurring over one or more *communication connections* as defined in IEC 61499-1, utilizing instances of the function block types defined in Annex D.

D.6.2 Service conventions

Except for the extensions defined in Annex D, the conventions for naming of input and output variables and events, and for describing the *services* (as defined in this standard) provided by instances of the function block types described in Annex D, are as defined in IEC 61499-1 for the descriptions of *service interface function block types* and *communication function block types*.

For the purposes of Annex D, the `PARAMS` input of type `ANY` defined in this standard is replaced by an `ID` input of type `WSTRING`. The contents of this string specify an **implementation-dependent** representation of the path to the *variable* of interest in the server.

EXAMPLE 1 In the case where the IEC 61499 client device is in logical proximity to the IEC 61131 server, it may be sufficient to simply name the IEC 61131-3 *access path* to the desired variable in the `ID` input, for instance `"CELL_1.CHARLIE"` in the example shown in Figure 19a of IEC 61131-3:2003.

EXAMPLE 2 In the case where the IEC 61499 client device is remotely connected to the IEC 61131-3 server via a communication network, it may be possible to use the `ID` input to encapsulate a Universal Resource Identifier (URI) to specify the desired access path, for instance, `"http://192.168.0.1:61131/CELL_1.CHARLIE"`.

NOTE Where supported by an implementation, the `ID` input may specify an access path to a status variable, such as the pre-defined access paths `P_PCSTATUS` and `P_PCSTATE` specified in IEC 61131-5.

Where used, the contents of the `TYPE` input of a function block type defined in Annex D specify the name of the *data type* of the data (`SD` or `RD`) being transferred. This may be the name of an elementary data type such as `"BOOL"` or a derived data type such as `"ANALOG_16_INPUT_DATA"`.

Where used, the contents of the `TASK` input of a function block type defined in Annex D specify an **implementation-dependent** representation of the path to the *task* of interest in the server.

EXAMPLE 3 In the case where an IEC 61499 client device is in logical proximity to an IEC 61131-3 server configured as shown in Figure 19a of IEC 61131-3:2003, a path to the task named `SLOW_1` in resource `STATION_1` could be represented as `"CELL_1.STATION_1.SLOW_1"`.

Values of the `STATUS` output of the function block types defined in D.6.3 are as given in Table 24 of IEC 61131-5:2000.

D.6.3 Function block types

D.6.3.1 READ

An instance of the `READ` function block type shown graphically in Figure D.2 and textually in Table D.2 can be used by an IEC 61499 client device to read program or status variable values from an IEC 61131-3 server.

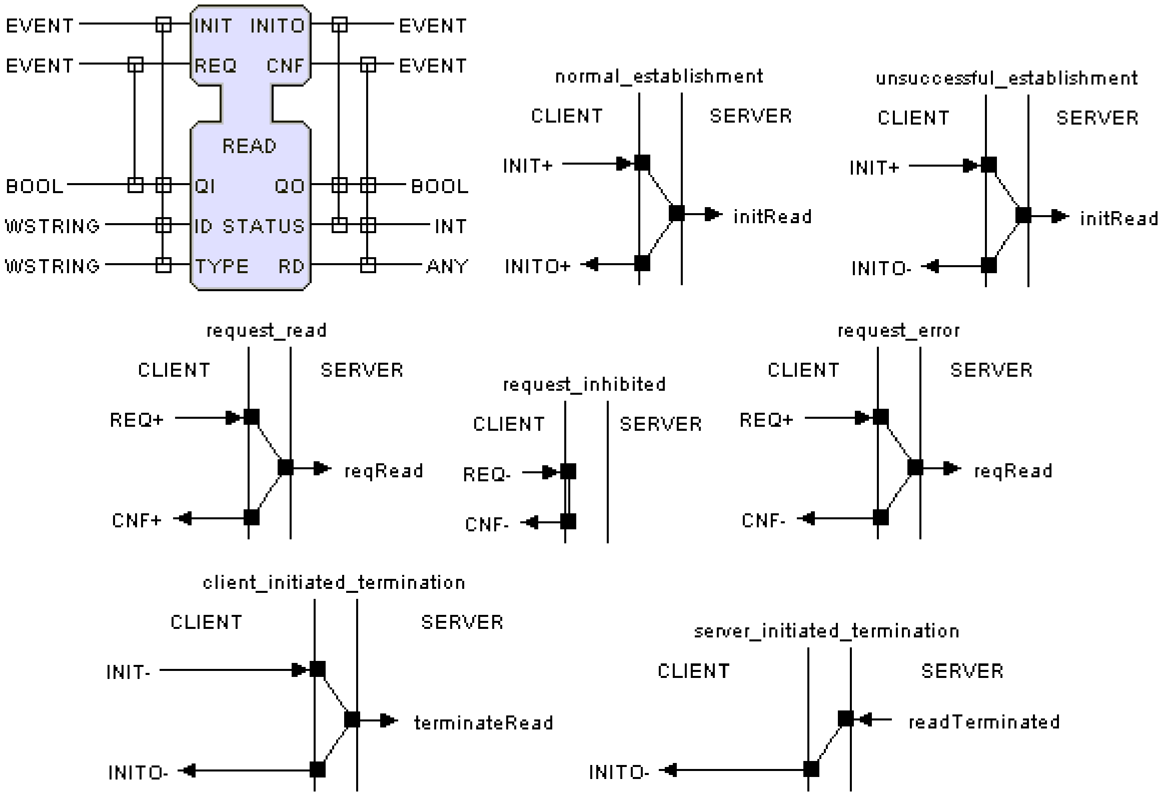


Figure D.2 – Function block type READ

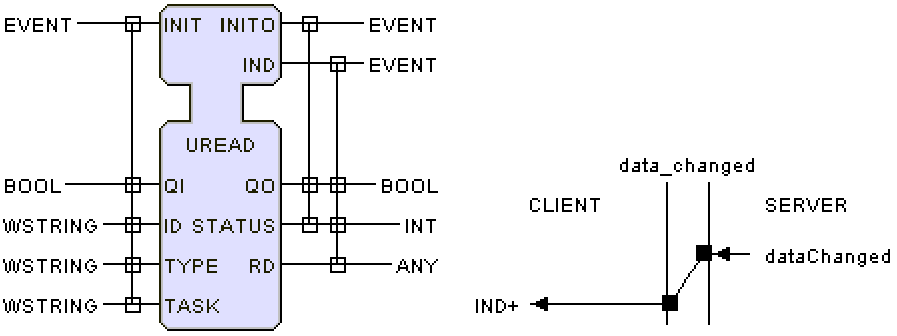
Table D.2 – Source code of function block type READ

```
FUNCTION_BLOCK READ (* Read server status or program variable *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialize/Terminate Service *)
  REQ WITH QI; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS,RD; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be read *)
  TYPE: WSTRING; (* Data type of RD variable *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
  RD: ANY; (* Variable data from IEC 61131 device *)
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initRead(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_read
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF+(RD);
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+() -> SERVER.reqRead(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateRead(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.readTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.2 UREAD

An instance of the `UREAD` function block type shown graphically in Figure D.3 and textually in Table D.3 can be used by an IEC 61499 client device to request asynchronous notification of a change in value of a program or status variable from an IEC 61131-3 server. Notification is received via the block's `IND` event output upon completion of the execution of the specified task when a change in the value of the specified variable (with respect to its value upon initiation of task execution) is detected.

An instance of this function block type can also be used to receive notification of the completion of each execution of the specified task by leaving unspecified the `ID` and `TYPE` inputs of the block.



NOTE The graphical representation of other service sequences listed in Table D.3 is similar to Figure D.2.

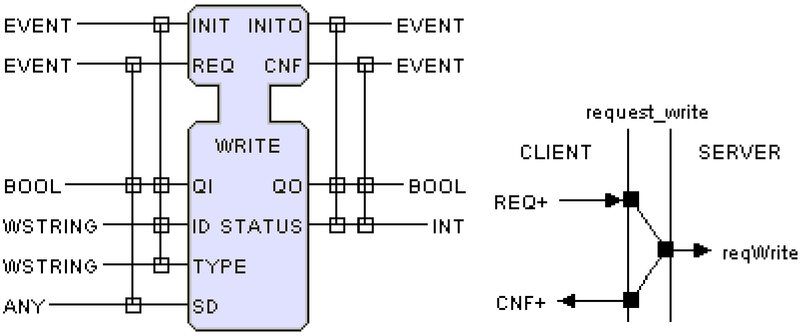
Figure D.3 – Function block type UREAD

Table D.3 – Source code of function block type UREAD

```
FUNCTION_BLOCK UREAD (* Unsolicited read of IEC 61131 program or status variable *)
EVENT_INPUT
  INIT WITH QI,ID,TASK,TYPE; (* Initialize/Terminate Service *)
END_EVENT
EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  IND WITH QO,STATUS,RD; (* Indication of changed RD value *)
END_EVENT
VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be read *)
  TYPE: WSTRING; (* Data type of RD variable *)
  TASK: WSTRING; (* Path to IEC 61131 TASK triggering read on changed value *)
END_VAR
VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
  RD: ANY; (* Input data from resource *)
END_VAR
SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK) -> CLIENT.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE,TASK) -> SERVER.initURead(ID,TYPE,TASK)
    -> CLIENT.INITO-(STATUS);
END_SEQUENCE
SEQUENCE data_changed
  SERVER.dataChanged() -> CLIENT.IND+(RD);
END_SEQUENCE
SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateURead() -> CLIENT.INITO-(STATUS);
END_SEQUENCE
SEQUENCE server_initiated_termination
  SERVER.UReadTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.3 WRITE

An instance of the `WRITE` function block type shown graphically in Figure D.4 and textually in Table D.4 can be used by an IEC 61499 client device to write variable data values to an IEC 61131-3 server.



NOTE The graphical representation of other service sequences listed in Table D.4 is similar to Figure D.2.

Figure D.4 – Function block type WRITE

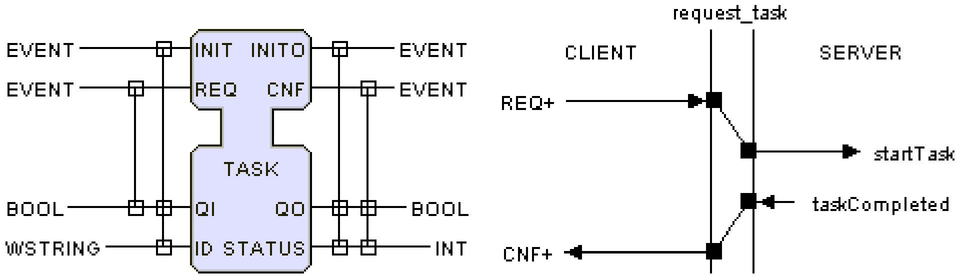
Table D.4 – Source code of function block type WRITE

```
FUNCTION_BLOCK WRITE (* Write a variable value to an IEC 61131 server *)
EVENT_INPUT
  INIT WITH QI,ID,TYPE; (* Initialize/Terminate Service *)
  REQ WITH QI,SD; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to variable to be written *)
  TYPE: WSTRING; (* Data type of SD variable *)
  SD: ANY; (* Variable value to write *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID,TYPE) -> SERVER.initWrite(ID,TYPE) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_write
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID,SD) -> SERVER.reqWrite(ID,SD) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateWrite(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.writeTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.3.4 TASK

An instance of the `TASK` function block type shown graphically in Figure D.5 and textually in Table D.5 can be used by an IEC 61499 client device to request the execution of a task on an IEC 61131-3 server.

When an implementation supports this feature, no value is configured for either the `SINGLE` or `INTERVAL` input of the corresponding TASK block as defined in Table 50 of IEC 61131-3:2003; rather, execution of the corresponding task is triggered as shown in the `request_task` service sequence shown in Figure D.5.



NOTE The graphical representation of other service sequences listed in Table D.5 is similar to Figure D.2.

Figure D.5 – Function block type TASK

Table D.5 – Source code of function block type TASK

```
FUNCTION_BLOCK TASK (* Trigger IEC 61131 task *)
EVENT_INPUT
  INIT WITH QI,ID; (* Initialize/Terminate Service *)
  REQ WITH QI; (* Service Request *)
END_EVENT

EVENT_OUTPUT
  INITO WITH QO,STATUS; (* Initialize/Terminate Confirm *)
  CNF WITH QO,STATUS; (* Confirmation of Requested Service *)
END_EVENT

VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  ID: WSTRING; (* Path to task to be triggered *)
END_VAR

VAR_OUTPUT
  QO: BOOL; (* 1=Normal operation, 0=Abnormal operation *)
  STATUS: INT;
END_VAR

SERVICE CLIENT/SERVER
SEQUENCE normal_establishment
  CLIENT.INIT+(ID) -> SERVER.initTask(ID) -> CLIENT.INITO+();
END_SEQUENCE

SEQUENCE unsuccessful_establishment
  CLIENT.INIT+(ID) -> SERVER.init(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE request_task
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF+();
END_SEQUENCE

SEQUENCE request_inhibited
  CLIENT.REQ-() -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE request_error
  CLIENT.REQ+(ID) -> SERVER.reqTask(ID) -> CLIENT.CNF-(STATUS);
END_SEQUENCE

SEQUENCE client_initiated_termination
  CLIENT.INIT-() -> SERVER.terminateTask(ID) -> CLIENT.INITO-(STATUS);
END_SEQUENCE

SEQUENCE server_initiated_termination
  SERVER.taskTerminated(ID,STATUS) -> CLIENT.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
```

D.6.4 Compliance

The specifications given in Annex D may be referenced in compliance profiles according to the rules given in IEC 61499-4.

When a programmable controller system compliant with IEC 61131-3 supports interoperability with one or more of the IEC 61499 function block types defined in Annex D, it should include in its list of supported features a reference to the supported features taken from Table D.6, and should include specifications of the values for implementation specific features and parameters as defined in 8.1 and 8.2 of IEC 61131-5:2000, respectively.

Table D.6 – IEC 61499 interoperability features

No.	Description
1	READ function block type
2	UREAD function block type
3	WRITE function block type
4	TASK function block type

Annex E
(informative)

Information exchange

E.1 Use of application layer facilities

Subclause 7.1.3.2 of ISO/IEC 7498-1:1994 identifies a number of facilities provided by *application-entities* (i.e., *entities* in the *application layer*) to enable *application-processes* to exchange information. To provide these facilities, the application-entities use *application-protocols* and *presentation services.* The communication function blocks defined in Clause E.2 may use these facilities, when provided by appropriate application-entities, in the following ways.

a) Communication function blocks utilize the *information transfer* facilities provided by application-entities to provide the *synchronization of cooperating applications* represented by the `REQ`, `CNF`, `IND`, and `RSP` events and to transfer the data represented by the `SD` inputs and `RD` outputs.
b) The following facilities may be used during service initialization as represented by the `INIT` and `INITO` events, using elements of the `PARAMS` data structure as necessary:
   - identification of the intended communications partners;
   - determination of the acceptable quality of service;
   - agreement on responsibility for error recovery;
   - agreement on security aspects;
   - identification of abstract syntax.
c) Facilities for *selection of mode of dialog* may be used by the specific function block types, e.g., by a `SUBSCRIBER` to ensure that it is interacting properly with a `PUBLISHER`.

Many of the facilities listed above may not be provided by application-entities of industrial-process measurement and control systems (IPMCSs). In this case, the communication function blocks shall implement equivalent facilities to provide the required services.

In particular, presentation services are often not provided by IPMCS application-entities. Therefore, in order to facilitate implementation of these services by communication function blocks, transfer syntaxes for both information transfer and application management are defined in Clause E.3.

NOTE 1 See ISO/IEC 7498-1 for definitions of terms used in this annex, but not defined in this standard.

NOTE 2 A *resource* is an "application-process" as defined in ISO/IEC 7498-1.

NOTE 3 The contents of Annex E could be considered normative in that compliance profiles as defined in IEC 61499-4, other standards and specifications can specify a context within which some or all of its provisions are employed.

E.2 Communication function block types

E.2.1 General

This subclause defines generic *communication function block types* for *unidirectional* and *bidirectional transactions.* **Implementation-dependent** customizations of these types should adhere to the following rules:

a) the implementation shall specify the data types and semantics of values of the data inputs and data outputs of each such function block type;

b) the implementation shall specify the treatment of abnormal data transfer;

c) the implementation shall specify any differences between the behavior of instances of such function block types and the behaviors specified in Clause E.2.

E.2.2 Function blocks for unidirectional transactions

Figures E.1 through E.4 provide type declarations and typical service primitive sequences of function blocks which provide *unidirectional transactions* over a *communication connection*. Such a connection consists of one *instance* of `PUBLISH` and one or more instances of `SUBSCRIBE` type.

NOTE 1 Full textual specifications of these function block types are not given in Annex F.

NOTE 2 The data types and semantics of the `PARAMS` input and `STATUS` output are **implementation-dependent**.

NOTE 3 The number (`m`) and types of the received data `RD_1,...,RD_m` correspond to the number and types of the transmitted data `SD_1,...,SD_m`.

NOTE 4 The means by which communication connections are set up are beyond the scope of this standard.

NOTE 5 Data transfer might be required in order to determine whether `RD_1,...,RD_m` meet the constraints expressed in Note 3.

NOTE 6 The transfer syntaxes defined in Clause E.3 can be used to make the determination described in Note 5.

NOTE 7 Treatment of abnormal data transfer is **implementation-dependent.**

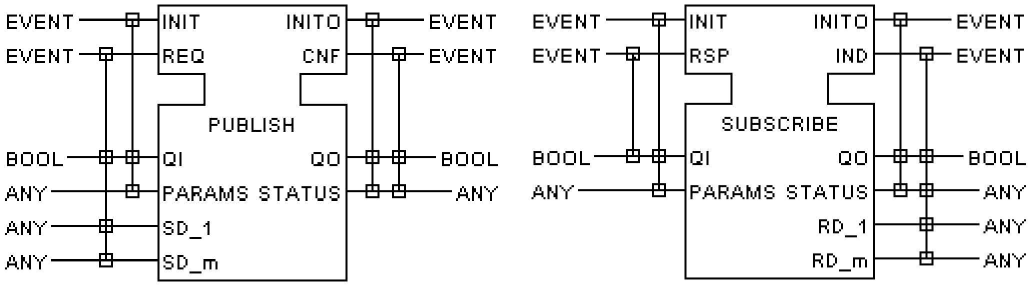


Figure E.1 – Type specifications for unidirectional transactions

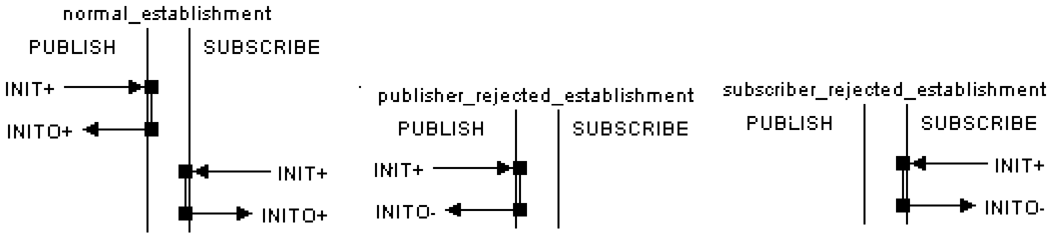


Figure E.2 – Connection establishment for unidirectional transactions

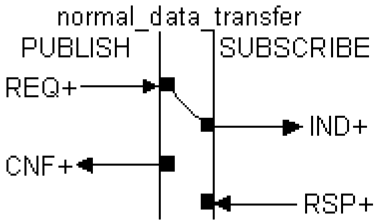


Figure E.3 – Normal unidirectional data transfer

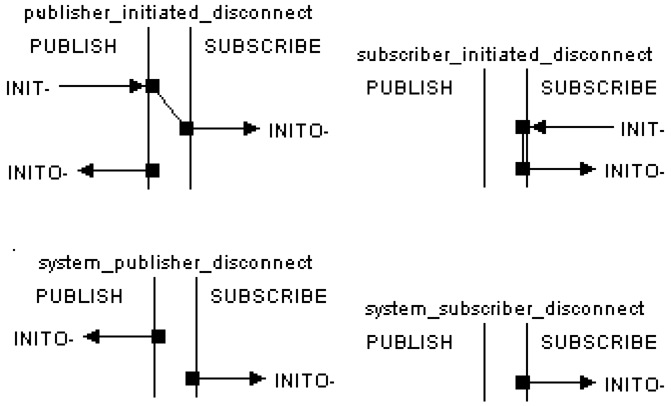


Figure E.4 – Connection release in unidirectional data transfer

E.2.3 Function blocks for bidirectional transactions

Figures E.5 through E.8 provide type declarations and service primitive sequences of function blocks which provide *bidirectional transactions* over a *communication connection*. Such a connection consists of one instance of CLIENT type and one instance of SERVER type.

NOTE 1 Full textual specifications of these function block types are not given in Annex F.

NOTE 2 The data types and semantics of the `PARAMS` input and `STATUS` output are **implementation-dependent**.

NOTE 3 The number (`m`) and types of the received data `RD_1,...,RD_m` correspond to the number and types of the transmitted data `SD_1,...,SD_m`.

NOTE 4 The number (`n`) and types of the received data `RD_1,...,RD_n` correspond to the number and types of the transmitted data `SD_1,...,SD_n`.

NOTE 5 Data transfer may be required in order to determine whether `RD_1,...,RD_m` and `RD_1,...,RD_n` meet the constraints expressed in Notes 3 and 4.

NOTE 6 The transfer syntaxes defined in Clause E.3 may be used to make the determination described in Note 5.

NOTE 7 Treatment of abnormal data transfer is **implementation-dependent.**

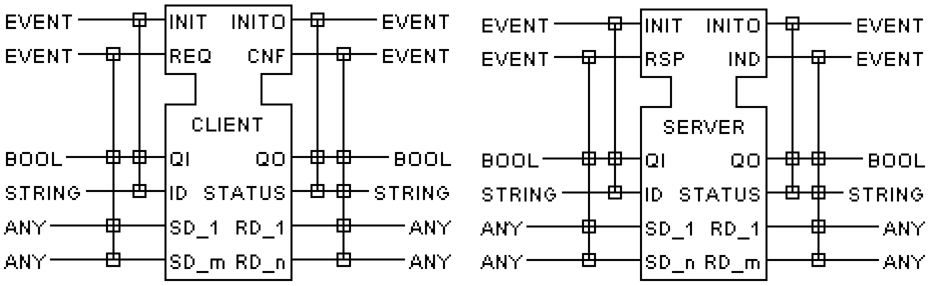


Figure E.5 – Type specifications for bidirectional transactions

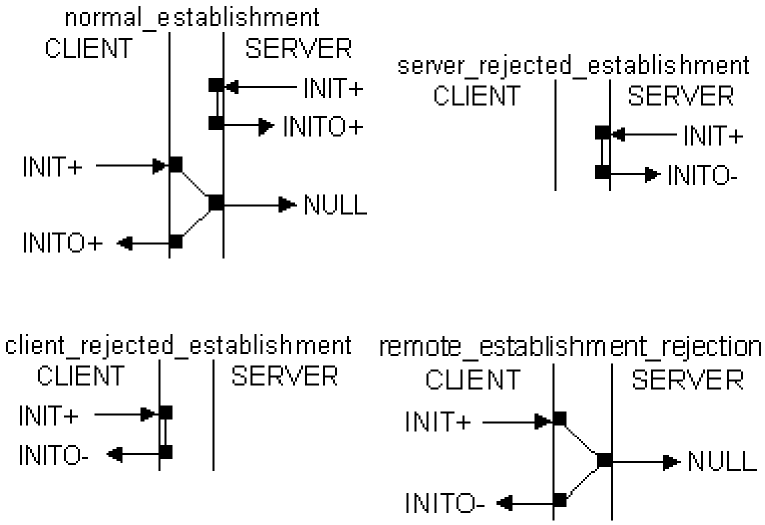


Figure E.6 – Connection establishment for bidirectional transaction

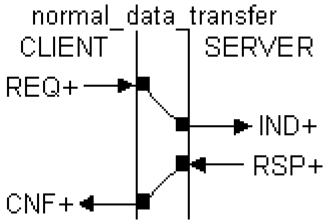


Figure E.7 – Bidirectional data transfer

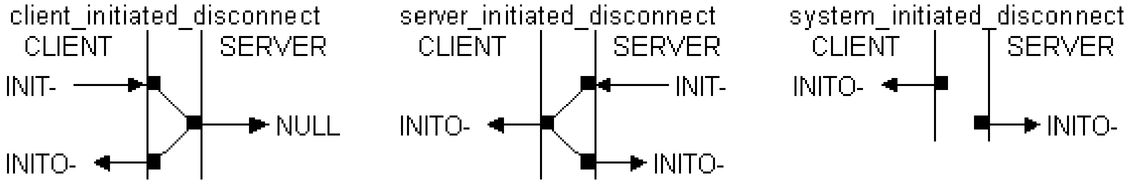


Figure E.8a – Client initiated

Figure E.8b – Server initiated

Figure E.8c – System initiated

Figure E.8 – Connection release in bidirectional data transfer

E.3 Transfer syntaxes

E.3.1 Background

A transfer syntax is defined in terms of an *abstract syntax* describing the types of data to be transferred, and a set of *encoding rules* for encoded representation of instances of the data types so defined. Subclause E.3.2 utilizes Abstract Syntax Notation One (ASN.1), as defined in ISO/IEC 8824-1, to define the IEC61499-FBDATA syntax for data transfer.

Two sets of encoding rules are given in Annex E:

a) Subclause E.3.3.1 defines BASIC encoding rules, utilizing the rules defined in ISO/IEC 8825-1.

b) Subclause E.3.3.2 utilizes the special characteristics of the data types in the IEC61499-FBDATA syntax to obtain a set of COMPACT encoding rules according to the following principles:
   - Where the number of "contents octets" is fixed, "length octets" are not used in the encoding.
   - Special encodings are used to minimize the number of octets and encoding/decoding effort required for fixed length types.
   - "Identifier octets" are not used for individual elements of STRUCT and ARRAY data types, since the type of each element is fixed in the corresponding *type declaration*.

E.3.2 IEC61499-FBDATA abstract syntax

The transfer syntax obtained by applying the COMPACT encoding rules in E.3.3.2 to the abstract syntax in E.3.2 is recommended for:

- transferring values from the `SD` inputs of a *communication function block* to the `RD` outputs of the communication function block(s) at the opposite end of a *communication connection*;
- determining whether the constraints on corresponding number and type of variables between `SD` inputs and `RD` outputs are met as noted in Figures E.1 and E.5.

The use of the abstract syntax defined in E.3.2 for the transfer of data expressed as *literals* and values of *variables* is subject to the following semantic RULES:

a) Where the name of a data type in this module (for example, `BOOL`) corresponds to the name of a data type defined in IEC 61131-3, the type definition given is intended for the transfer of data of the corresponding IEC 61131-3 data type.

b) The values of "VisibleString" for the data types `DATE` and `TIME_OF_DAY` is restricted to the textual syntax for these data types as defined in IEC 61131-3.

c) The notation `[typeID]` implies that the tag of the data consists of the value of the `ASN.1 tag` of the corresponding derived data type, established as specified in Annex A of IEC 61499-2:2005 or by other means beyond the scope of this standard.

d) The value of an `EnumeratedData` item consists of the cardinal position (beginning at zero) of the corresponding identifier in the sequence of identifiers defined for the corresponding enumerated data type, established as specified in IEC 61131-3.

e) The specific type of a `SubrangeData` item is as for its particular *subrange data type,* declared as specified in IEC 61131-3.

f) The type of the elements of an `ARRAY` data item is established as specified for *array data types* in IEC 61131-3.

g) The types of the elements of a `STRUCT` data item are established as specified for *structured data types* in IEC 61131-3.

ASN.1 MODULE

```
IEC61499-FBDATA DEFINITIONS::=

BEGIN

EXPORTS FBDataSequence, FBData, ElementaryData, BOOL, FixedLengthInteger,
     FixedLengthReal, TIME, AnyDate, AnyString, FixedLengthBitString,
     SignedInteger, UnsignedInteger, REAL, LREAL, DATE, TIME_OF_DAY,
     DATE_AND_TIME, STRING, WSTRING, BYTE, WORD, DWORD, LWORD,
     DirectlyDerivedData, EnumeratedData, SubrangeData, ARRAY, STRUCT;

FBDataSequence::= [APPLICATION 23] IMPLICIT SEQUENCE OF FBData

FBData::= CHOICE{ElementaryData, DerivedData}
```

```
ElementaryData::= CHOICE{
     BOOL,
     FixedLengthInteger,
     FixedLengthReal,
     TIME,
     AnyDate,
     AnyString,
     FixedLengthBitString}

FixedLengthInteger::= CHOICE{SignedInteger, UnsignedInteger}

SignedInteger::= CHOICE{SINT, INT, DINT, LINT}

UnsignedInteger::= CHOICE{USINT, UINT, UDINT, ULINT}

FixedLengthReal::= CHOICE{REAL, LREAL}

AnyDate::= CHOICE{DATE, TIME_OF_DAY, DATE_AND_TIME}

AnyString::= CHOICE{STRING, WSTRING}

FixedLengthBitString::= CHOICE{BYTE, WORD, DWORD, LWORD}

BOOL::= CHOICE{BOOL0, BOOL1}

BOOL0::= [APPLICATION 0] IMPLICIT NULL

BOOL1::= [APPLICATION 1] IMPLICIT NULL

SINT::= [APPLICATION 2] IMPLICIT INTEGER(-128..127)

INT::= [APPLICATION 3] IMPLICIT INTEGER(-32768..32767)

DINT::= [APPLICATION 4] IMPLICIT INTEGER(-2147483648..2147483647)

LINT::= [APPLICATION 5]
     IMPLICIT INTEGER(-9223372036854775808..9223372036854775807)

USINT::= [APPLICATION 6] IMPLICIT INTEGER(0..255)

UINT::= [APPLICATION 7] IMPLICIT INTEGER(0..65535)

UDINT::= [APPLICATION 8] IMPLICIT INTEGER(0..4294967295)

ULINT::= [APPLICATION 9] IMPLICIT INTEGER(0..18446744073709551615)

REAL::= [APPLICATION 10] IMPLICIT OCTET STRING (SIZE(4))

LREAL::= [APPLICATION 11] IMPLICIT OCTET STRING (SIZE(8))

TIME::= [APPLICATION 12] IMPLICIT LINT -- Duration in 1µs units

DATE::= [APPLICATION 13] IMPLICIT ULINT -- See Table E.1.

TIME_OF_DAY::= [APPLICATION 14] IMPLICIT ULINT -- See Table E.1.

DATE_AND_TIME::= [APPLICATION 15] IMPLICIT ULINT -- See Table E.1.

STRING::= [APPLICATION 16] IMPLICIT OCTET STRING -- 1 octet/char

BYTE::= [APPLICATION 17] IMPLICIT BIT STRING (SIZE(8))

WORD::= [APPLICATION 18] IMPLICIT BIT STRING (SIZE(16))

DWORD::= [APPLICATION 19] IMPLICIT BIT STRING (SIZE(32))

LWORD::= [APPLICATION 20] IMPLICIT BIT STRING (SIZE(64))

WSTRING::= [APPLICATION 21] IMPLICIT OCTET STRING -- 2 octets/char

DerivedData::= CHOICE{
     DirectlyDerivedData,
     EnumeratedData,
     SubrangeData,
     ARRAY,
     STRUCT}
```

```
DirectlyDerivedData::= [typeID] IMPLICIT ElementaryData

EnumeratedData::= [typeID] IMPLICIT UINT

SubrangeData::= [typeID] IMPLICIT FixedLengthInteger

ARRAY::= CHOICE {ArrayVariable, TypedArray}

ArrayVariable::= [APPLICATION 22] IMPLICIT FBDataSequence -- same type

TypedArray::= [typeID] IMPLICIT FBDataSequence - same type

STRUCT::= [typeID] IMPLICIT SEQUENCE -- different types

END
```

E.3.3 Encoding rules

E.3.3.1 BASIC encoding

This encoding shall be the result of applying the basic encoding rules of ISO/IEC 8825-1 to variables of the types defined in E.3.2.

E.3.3.2 COMPACT encoding

This encoding shall be the result of modifying the rules for BASIC encoding given in E.3.3.1 as follows.

a) "Length octets" shall not be included in the encoding of values of the data types shown in Table E.1.

b) The length (in octets) and encoding of the "contents octets" described in ISO/IEC 8825-1 shall be as defined in Table E.1 for values of the data types shown there.

c) Encoding of variables of `TIME`, `DirectlyDerivedData`, `EnumeratedData`, or `SubrangeData` types shall follow the same encoding rules as the base type.

d) "Type octets" shall not be included in the encoding of individual elements of `STRUCT` types, except for the encoding of elements of type `BOOL`, which shall be encoded according to rule (1) of Table E.1.

e) The encoding of values of `STRING` and `WSTRING` types shall be primitive.

f) The encoding of `ARRAY` elements shall be *constructed* in the sense of ISO/IEC 8825-1, with the following provisions for COMPACT encoding:

    1) The "length" subfield of the `ARRAY` element shall be encoded as a value of the `UINT` type without identifier or length octets, i.e., as a 16-bit unsigned integer;

       NOTE 1 This would appear to restrict the maximum number of elements of an `ARRAY` to 65535. However, the actual length may be further restricted by the maximum number of octets that can be transferred by the underlying transport protocol.

       EXAMPLE For UDP messages with a maximum number of is 65508 octets, the maximum transmittable length of an `ARRAY` of `BYTE` elements would be (maximum octets - tag octets - length octets - element type octets)/(element length) = (65508-1-2-1)/1 = 65504 elements.

    2) COMPACT encoding shall be used for the first element of the "values" field;

    3) Subsequent elements, if any, shall be encoded using the COMPACT syntax without an "identifier" subfield, except for elements of type `BOOL`, which shall be encoded according to rule (1) of Table E.1;

    4) If the specified length of the received `ARRAY` is less than the locally allocated space, the remaining elements of the local array are unaffected; if the length of the received `ARRAY` is greater than the locally allocated space, the remaining received elements are ignored.

NOTE 2 Since `ARRAY` is a subclass of `FBData`, a multidimensional `ARRAY` can be encoded recursively as an `ARRAY` whose elements are `ARRAY` elements.

Table E.1 – COMPACT encoding of fixed length data types

Data type	Contents octets	
	Length	Encoding rule
`BOOL`	0	(1)
`SINT`	1	(2)
`INT`	2	(2)
`DINT`	4	(2)
`LINT`	8	(2)
`USINT`	1	(3)
`UINT`	2	(3)
`UDINT`	4	(3)
`ULINT`	8	(3)
`REAL`	4	(4)
`LREAL`	8	(4)
`DATE`	8	(5)
`TIME`	8	(7)
`TIME_OF_DAY`	12	(5)
`DATE_AND_TIME`	20	(5)
`BYTE`	1	(6)
`WORD`	2	(6)
`DWORD`	4	(6)
`LWORD`	8	(6)

ENCODING RULES FOR TABLE E.1

(1) Values of this data type shall be encoded as a single identifier octet containing the tag encoding for the `BOOL0` or `BOOL1` class, as defined in E.3.2, corresponding to values of `FALSE (0)` or `TRUE (1)`, respectively.

(2) Values of these `SignedInteger` data types shall be encoded in the same manner as an `UnsignedInteger` of the same length as the `SignedInteger` type with a value of $N - N_{min}$, where N is the value of the `SignedInteger` variable to be encoded and N_{min} is the lower end point of the value range of the `SignedInteger` subtype as defined in E.3.2.

(3) Values of these `UnsignedInteger` data types shall be encoded by numbering the bits in the contents octets, starting with bit 1 of the last octet as bit zero and ending the numbering with bit 8 of the first octet. Each bit is assigned a value of 2^N, where N is its position in the above numbering sequence. The value of the unsigned integer is obtained by summing the numerical values assigned to each bit for those bits which are set to one.

(4) Values of these data types shall be encoded as 32-bit single format and 64-bit double format numbers, respectively, as defined in ISO/IEC/IEEE 60559, where the "lsb" defined in ISO/IEC/IEEE 60559 corresponds to "bit zero" as defined in Rule (3).

(5) Values of these types shall be encoded as for type `ULINT`, representing the number of milliseconds since midnight for `TIME_OF_DAY`, the number of milliseconds since `1970-01-01-00:00:00.000` for `DATE_AND_TIME`, or the number of milliseconds from `1970-01-01-00:00:00.000` to `YYYY-MM-DD-00:00:00.000` for `DATE`, where `YYYY-MM_DD` is the current date.

(6) Encoding of values of these `FixedLengthBitString` data types shall be primitive, and shall be obtained by placing the bits in the bitstring, commencing with the first bit and proceeding to the trailing bit, in bits 8 to 1 of the first contents octet, followed in turn by bits 8 to 1 of each of the subsequent octets, where the notation "first bit" and "trailing bit" is specified in ISO/IEC 8824-1.

(7) Encoding of values of this data type shall be the same as for values of type `LINT`, representing a time interval in units of 1 μs.

Annex F
(normative)

Textual specifications

Annex F provides textual specifications, in the syntax defined in Annex B, for all function block and adapter types illustrated in this standard. The contents of Annex F are normative to the extent defined in the description of each such function block type or adapter type in this standard.

NOTE The specifications are listed alphabetically by type name.

```
====================================================================
FUNCTION_BLOCK E_CTU (* Event-Driven Up Counter *)
EVENT_INPUT
  CU WITH PV; (* Count Up *)
  R; (* Reset *)
END_EVENT
EVENT_OUTPUT
  CUO WITH Q,CV; (* Count Up Output Event *)
  RO WITH Q,CV; (* Reset Output Event *)
END_EVENT
VAR_INPUT
  PV: UINT; (* Preset Value *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* CV>=PV *)
  CV: UINT;
END_VAR
EC_STATES
  START;
  CU: CU -> CUO;
  R: R -> RO;
END_STATES
EC_TRANSITIONS
  START TO CU:= CU [CV<65535];
  CU TO START:= 1;
  START TO R:= R;
  R TO START:= 1;
END_TRANSITIONS
ALGORITHM CU IN ST: (* Count Up *)
CV:= CV + 1;
Q:= (CV >= PV);
END_ALGORITHM
ALGORITHM R IN ST: (* Reset *)
CV:= 0;
Q:= FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_CYCLE (* Periodic (cyclic) Generation of an Event *)
EVENT_INPUT
  START WITH DT;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO;   (* Periodic event at period DT, starting at DT after GO *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Period between events *)
END_VAR
FBS
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO DLY.START;
```

```
 STOP TO DLY.STOP;
 DLY.EO TO DLY.START;
 DLY.EO TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
 DT TO DLY.DT;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_D_FF (* Event-driven Data(D)Latch *)
EVENT_INPUT
  CLK WITH D; (* Data Clock *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Output Event when Q output changes *)
END_EVENT
VAR_INPUT
  D: BOOL; (* Data Input *)
END_VAR
VAR_OUTPUT
  Q: BOOL; (* Latched Data *)
END_VAR
EC_STATES
  Q0; (* Q is FALSE initially *)
  RESET: LATCH -> EO; (* Reset Q and issue EO *)
  SET: LATCH -> EO; (* Latch and issue EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= CLK [D];
  SET TO RESET:= CLK [NOT D];
  RESET TO SET:= CLK [D];
END_TRANSITIONS
ALGORITHM LATCH IN ST:
Q:=D;
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_DELAY
 (* Delayed propagation of an event - Cancellable *)
EVENT_INPUT
 START WITH DT; (* Begin Delay *)
 STOP;      (* Cancel Delay *)
END_EVENT
EVENT_OUTPUT
 EO;     (* Delayed Event *)
END_EVENT
VAR_INPUT
 DT: TIME; (* Delay Time *)
END_VAR
SERVICE E_DELAY/RESOURCE
SEQUENCE event_delay
 E_DELAY.START(DT) ->E_DELAY.EO();
END_SEQUENCE
SEQUENCE delay_canceled
 E_DELAY.START(DT);
 E_DELAY.STOP();
END_SEQUENCE
SEQUENCE no_multiple_delay
 E_DELAY.START(DT);
 E_DELAY.START(DT);
 ->E_DELAY.EO();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_DEMUX (* Event demultiplexer *)
EVENT_INPUT
 EI WITH K;  (* Event to demultiplex *)
END_EVENT
```

```
EVENT_OUTPUT
 EO0;
 EO1;
 EO2;
 EO3;     (* Number of outputs is implementation dependent *)
END_EVENT
VAR_INPUT
 K: UINT;  (* Event index, maximum is implementation dependent *)
END_VAR
EC_STATES
 START; (* Initial State *)
 TRIGGERED; (* Intermediate state after EI arrives *)
 EO0: -> EO0;
 EO1: -> EO1;
 EO2: -> EO2;
 EO3: -> EO3;
END_STATES
EC_TRANSITIONS
 START TO TRIGGERED:= EI;
 TRIGGERED TO EO0:= [K=0];
 TRIGGERED TO EO1:= [K=1];
 TRIGGERED TO EO2:= [K=2];
 TRIGGERED TO EO3:= [K=3];
 TRIGGERED TO START:= [K>3];
 EO0 TO START:= 1;
 EO1 TO START:= 1;
 EO2 TO START:= 1;
 EO3 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_F_TRIG (* Boolean falling edge detection *)
EVENT_INPUT
 EI WITH QI;   (* Event Input *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Event Output *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Boolean input for falling edge detection *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO0 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_MERGE (* Merge (OR) of multiple events *)
EVENT_INPUT
  EI1; (* First input event *)
  EI2; (* Second input event *)
  END_EVENT
EVENT_OUTPUT EO; (* Output Event *)
END_EVENT
EC_STATES
  START;  (* Initial State *)
  EO:    (* Issue EO Event *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI1;
```

```
  START TO EO:= EI2;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
===================================================================
FUNCTION_BLOCK E_N_TABLE (* Generation of a finite train of separate events,
table driven *)
EVENT_INPUT
  START WITH DT, N;
  STOP;
END_EVENT
EVENT_OUTPUT
 EO0;   (* N events at periods DT, starting at DT[0] after START *)
 EO1;
 EO2;
 EO3;   (* Extensible *)
END_EVENT
VAR_INPUT
  DT: TIME[3]; (* Periods between events *)
  N: UINT;  (* Number of events to generate (=3 in this example) *)
END_VAR
SERVICE E_N_TABLE/RESOURCE
SEQUENCE typical_operation
  E_N_TABLE.START(DT,N) -> E_N_TABLE.EO0() -> E_N_TABLE.EO1() ->
E_N_TABLE.EO2() -> E_N_TABLE.EO3();
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
===================================================================
FUNCTION_BLOCK E_PERMIT (* Permissive propagation of an event *)
EVENT_INPUT EI WITH PERMIT; (* Event input *)
END_EVENT
EVENT_OUTPUT EO; (* Event output *)
END_EVENT
VAR_INPUT PERMIT: BOOL; END_VAR
EC_STATES
  START;  (* Initial State *)
  EO:    (* Issue EO Event *)
    ->EO;
END_STATES
EC_TRANSITIONS
  START TO EO:= EI [PERMIT];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
===================================================================
FUNCTION_BLOCK E_R_TRIG (* Boolean rising edge detection *)
EVENT_INPUT
 EI WITH QI;   (* Event Input *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Event Output *)
END_EVENT
VAR_INPUT
 QI: BOOL;  (* Boolean input for rising edge detection *)
END_VAR
FBS
 D: E_D_FF;
 SW: E_SWITCH;
END_FBS
EVENT_CONNECTIONS
  EI TO D.CLK;
  D.EO TO SW.EI;
  SW.EO1 TO EO;
END_CONNECTIONS
DATA_CONNECTIONS
  QI TO D.D;
  D.Q TO SW.G;
END_CONNECTIONS
```

```
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_REND (* Rendezvous of two events *)
EVENT_INPUT
  EI1; (* First Event Input *)
  EI2; (* Second Event Input *)
  R;  (* Reset Event *)
END_EVENT
EVENT_OUTPUT
 EO;  (* Rendezvous Output Event *)
END_EVENT
EC_STATES
  START;  (* Initial State *)
  EI1;   (* EI1 has arrived, wait for EI2 or R *)
  EO:    (* Issue rendezvous event *)
    ->EO;
  EI2;   (* EI2 has arrived, wait for EI1 or R *)
END_STATES
EC_TRANSITIONS
  START TO EI1:= EI1;
  EI1 TO START:= R;
  START TO EI2:= EI2;
  EI2 TO START:= R;
  EI1 TO EO:= EI2;
  EI2 TO EO:= EI1;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_RESTART (* Generation of Restart Events *)
EVENT_OUTPUT
 COLD;     (* Cold Restart *)
 WARM;     (* Warm Restart *)
END_EVENT
SERVICE RESOURCE/E_RESTART
SEQUENCE cold_restart ->E_RESTART.COLD(); END_SEQUENCE
SEQUENCE warm_restart ->E_RESTART.WARM(); END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_RS (* Event-driven bistable *)
EVENT_INPUT
  S; (* Set Event *)
  R; (* Reset Event *)
END_EVENT
EVENT_OUTPUT
  EO WITH Q; (* Output Event *)
END_EVENT
VAR_OUTPUT
  Q: BOOL; (* Current Output State *)
END_VAR
EC_STATES
  Q0; (* Q is FALSE initially *)
  RESET: RESET -> EO; (* Reset Q and issue EO *)
  SET: SET -> EO; (* Set Q and issue EO *)
END_STATES
EC_TRANSITIONS
  Q0 TO SET:= S;
  SET TO RESET:= R;
  RESET TO SET:= S;
END_TRANSITIONS
ALGORITHM SET IN ST: (* Set Q *)
Q:=TRUE;
END_ALGORITHM
ALGORITHM RESET IN ST: (* Reset Q *)
Q:=FALSE;
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_SELECT (* Selection between two events *)
```

```
EVENT_INPUT
  EI0 WITH G; (* Input event, selected when G=0 *)
  EI1 WITH G; (* Input event, selected when G=1 *)
END_EVENT
EVENT_OUTPUT EO; (* Output Event *)
END_EVENT
VAR_INPUT G: BOOL; (* Select EI0 when G=0, EI1 when G=1 *)
END_VAR
EC_STATES
  START;   (* Initial State *)
  EO: -> EO; (* Issue Output Event *)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI0 [NOT G];
  START TO EO:= EI1 [G];
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_SPLIT (* Split an event *)
EVENT_INPUT
  EI; (* Input event *)
END_EVENT
EVENT_OUTPUT
 EO1;  (* First output event *)
 EO2;  (* Second output event, etc. *)
END_EVENT
EC_STATES
  START; (* Initial State *)
  EO: (* Extensible *)
    ->EO1, (* Output first event *)
    ->EO2; (* Output second event, etc. *)
END_STATES
EC_TRANSITIONS
  START TO EO:= EI;
  EO TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_SWITCH (* Switch (demultiplex) an event *)
EVENT_INPUT EI WITH G; (* Event Input *)
END_EVENT
EVENT_OUTPUT
  EO0; (* Output, switched from EI when G=0 *)
  EO1; (* Output, switched from EI when G=1 *)
END_EVENT
VAR_INPUT G: BOOL; (* Switch EI to EI0 when G=0, to EI1 when G=1 *)
END_VAR
EC_STATES
  START; (* Initial State *)
  G0:  (* Issue EO0 when EI arrives with G=0 *)
    ->EO0;
  G1: (* Issue EO1 when EI arrives with G=1 *)
    ->EO1;
END_STATES
EC_TRANSITIONS
  START TO G0:= EI [NOT G];
  G0 TO START:= 1;
  START TO G1:= EI [G];
  G1 TO START:= 1;
END_TRANSITIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_TABLE (* Generation of a finite train of events, table
driven *)
EVENT_INPUT
 START WITH DT, N;
 STOP; (* Cancel *)
END_EVENT
```

```
EVENT_OUTPUT
 EO WITH CV; (* N events at periods DT, starting at DT[0] after START *)
END_EVENT
VAR_INPUT
  DT: TIME[4]; (* Periods between events *)
  N: UINT; (* Number of events to generate *)
END_VAR
VAR_OUTPUT
  CV: UINT; (* Current event index, 0..N-1 *)
END_VAR
FBS
  CTRL: E_TABLE_CTRL;
  DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
  START TO CTRL.INIT;
  CTRL.CLKO TO DLY.START;
  DLY.EO TO EO;
  DLY.EO TO CTRL.CLK;
  STOP TO DLY.STOP;
END_CONNECTIONS
DATA_CONNECTIONS
  DT TO CTRL.DT;
  N TO CTRL.N;
  CTRL.DTO TO DLY.DT;
  CTRL.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_TABLE_CTRL (* Control for E_TABLE *)
EVENT_INPUT
 INIT WITH DT, N;
 CLK;
END_EVENT
EVENT_OUTPUT
 CLKO WITH DTO, CV;
END_EVENT
VAR_INPUT
 DT: TIME[4]; (* Array length is implementation dependent *)
 N: UINT;   (* Actual number of time steps *)
END_VAR
VAR_OUTPUT
 DTO: TIME;  (* Current delay interval *)
 CV: UINT;   (* Current event index, 0..N-1 *)
END_VAR
EC_STATES
 START;
 INIT0: INIT;
 INIT1: -> CLKO;
 STEP: STEP -> CLKO;
END_STATES
EC_TRANSITIONS
 START TO INIT0:= INIT;
 INIT0 TO INIT1:= [N>0];
 INIT0 TO START:= [N=0]; (* Don't run if N=0 *)
 INIT1 TO START:= 1;
 START TO STEP:= CLK [CV < MIN(3,N-1)];
 STEP TO START:= 1;
END_TRANSITIONS
ALGORITHM STEP IN ST:
  CV:= CV+1;
  DTO:= DT[CV];
END_ALGORITHM
ALGORITHM INIT IN ST:
  CV:= 0;
  DTO:= DT[0];
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK E_TRAIN (* Generation of a finite train of events *)
```

```
EVENT_INPUT
 START WITH DT, N;
 STOP;
END_EVENT
EVENT_OUTPUT
 EO WITH CV; (* N events at period DT, starting at DT after START *)
END_EVENT
VAR_INPUT
  DT: TIME; (* Period between events *)
  N: UINT; (* Number of events to generate *)
END_VAR
VAR_OUTPUT
 CV: UINT; (* EO index (0..N-1) *)
END_VAR
FBS
 CTR: E_CTU;
 GATE: E_SWITCH;
 DLY: E_DELAY;
END_FBS
EVENT_CONNECTIONS
 START TO CTR.R;
 STOP TO DLY.STOP;
 DLY.EO TO EO;
 DLY.EO TO CTR.CU;
 CTR.CUO TO GATE.EI;
 CTR.RO TO GATE.EI;
 GATE.EO0 TO DLY.START;
END_CONNECTIONS
DATA_CONNECTIONS
 DT TO DLY.DT;
 N TO CTR.PV;
 CTR.Q TO GATE.G;
 CTR.CV TO CV;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK FB_ADD_INT (* INT Addition *)
EVENT_INPUT
 REQ WITH QI, IN1, IN2;
END_EVENT
EVENT_OUTPUT
 CNF WITH QO, STATUS, OUT;
END_EVENT
VAR_INPUT
 QI: BOOL;   (* Event Qualifier *)
 IN1: INT;   (* Augend *)
 IN2: INT;   (* Addend *)
END_VAR
VAR_OUTPUT
 QO: BOOL;    (* Output Qualifier *)
 STATUS: UINT;  (* Operation Status *)
 OUT: INT;   (* Sum *)
END_VAR
VAR
 RESULT: DINT;
END_VAR
EC_STATES
 START;
 REQ: REQ -> CNF;
END_STATES
EC_TRANSITIONS
 START TO REQ:= REQ;
 REQ TO START:= 1;
END_TRANSITIONS
ALGORITHM REQ IN ST:
  QO:= QI;
  IF QI THEN
   STATUS:= 0;
   RESULT:= INT_TO_DINT(IN1) + INT_TO_DINT(IN2);
```

```
   IF (RESULT > 32767) OR (RESULT < -32768) THEN
    QO = FALSE;
    STATUS = 3;
    IF (RESULT > 32767) THEN OUT:= 32767;
    ELSE OUT:= -32768;
    END_IF;
   ELSE OUT:= RESULT;
   END_IF;
  ELSE STATUS = 1;
  END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK INTEGRAL_REAL
EVENT_INPUT
 INIT: INIT_EVENT WITH CYCLE;
 EX WITH HOLD, XIN;
END_EVENT
EVENT_OUTPUT
 INITO: INIT_EVENT WITH XOUT;
 EXO WITH XOUT;
END_EVENT
VAR_INPUT
 HOLD: BOOL;  (* 0 = Run, 1 = Hold *)
 XIN: REAL;  (* Integrand *)
 CYCLE: TIME; (* Sampling period *)
END_VAR
VAR_OUTPUT
 XOUT: REAL;  (* Integrated output *)
END_VAR
VAR DT: REAL; END_VAR
EC_STATES
 START;        (* EC Initial state *)
 INIT:INIT -> INITO; (* EC State with Algorithm and EC Action *)
 MAIN: MAIN -> EXO;
END_STATES
EC_TRANSITIONS
 START TO INIT:= INIT; (* An EC Transition *)
 START TO MAIN:= EX;
 INIT TO START:= 1;
 MAIN TO START:= 1;
END_TRANSITIONS
ALGORITHM INIT IN ST:
 XOUT:= 0.0;
 DT:= TIME_TO_REAL(CYCLE);
END_ALGORITHM
ALGORITHM MAIN IN ST:
 IF NOT HOLD THEN
  XOUT:= XOUT + XIN * DT;
 END_IF;
END_ALGORITHM
END_FUNCTION_BLOCK
====================================================================
ADAPTER LD_UNLD (* LOAD/UNLOAD Adapter Interface *)
EVENT_INPUT
 UNLD;    (* UNLOAD Request *)
END_EVENT
EVENT_OUTPUT
 LD WITH WO,WKPC; (* LOAD Request *)
 CNF WITH WO,WKPC; (* UNLD Confirm *)
END_EVENT
VAR_OUTPUT
 WO: BOOL;  (* Workpiece present *)
 WKPC: COLOR; (* Workpiece Color *)
END_VAR
SERVICE PLUG/SOCKET
SEQUENCE normal_operation
 PLUG.LD(WO,WKPC) -> SOCKET.LD(WO,WKPC);
 SOCKET.UNLD() -> PLUG.UNLD();
 PLUG.CNF() -> SOCKET.CNF();
```

```
END_SEQUENCE
END_SERVICE
END_ADAPTER
===============================================================
FUNCTION_BLOCK MANAGER (* Management Service Interface *)
EVENT_INPUT
  INIT WITH QI, PARAMS; (* Service Initialization *)
  REQ WITH QI, CMD, OBJECT; (* Service Request *)
END_EVENT
EVENT_OUTPUT
  INITO WITH QO, STATUS; (* Initialization Confirm *)
  CNF WITH QO, STATUS, RESULT; (* Service Confirmation *)
END_EVENT
VAR_INPUT
  QI: BOOL; (* Event Input Qualifier *)
  PARAMS: WSTRING; (* Service Parameters *)
  CMD: UINT; (* Enumerated Command *)
  OBJECT: BYTE[512]; (* Command Object *)
END_VAR
VAR_OUTPUT
  QO: BOOL; (* Event Output Qualifier *)
  STATUS: UINT; (* Service Status *)
  RESULT: BYTE[512]; (* Result Object *)
END_VAR
SERVICE MANAGER/resource
SEQUENCE normal_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement() -> MANAGER.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
  MANAGER.INIT+(PARAMS) -> resource.initManagement(PARAMS) -> MANAGER.INITO-
(STATUS);
END_SEQUENCE
SEQUENCE normal_command_sequence
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.CNF+(STATUS,RESULT);
END_SEQUENCE
SEQUENCE command_error
  MANAGER.REQ+(CMD,OBJECT) -> resource.performCommand(CMD,OBJECT) ->
MANAGER.IND-(STATUS);
END_SEQUENCE
SEQUENCE application_initiated_termination
  MANAGER.INIT-() -> resource.terminateService() -> MANAGER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE resource_initiated_termination
  resource.serviceTerminated(STATUS) -> MANAGER.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK PI_REAL
EVENT_INPUT
  INIT WITH KP, KI, CYCLE;
  EX WITH HOLD, PV, SP, KP, KI, CYCLE;
END_EVENT
EVENT_OUTPUT
  INITO WITH XOUT;
  EXO WITH XOUT;
END_EVENT
VAR_INPUT
  HOLD: BOOL; (* Hold when TRUE *)
  PV: REAL; (* Process variable *)
  SP: REAL; (* Set point *)
  KP: REAL; (* Proportionality constant *)
  KI: REAL; (* Integral constant,1/s *)
  CYCLE: TIME; (* Sampling period *)
END_VAR
VAR_OUTPUT
  XOUT: REAL;
END_VAR
```

```
FBS
  CALC: PID_CALC;
  INTEGRAL_TERM: INTEGRAL_REAL;
END_FBS
EVENT_CONNECTIONS
  INIT TO CALC.INIT;
  EX TO CALC.PRE;
  CALC.POSTO TO EXO;
  INTEGRAL_TERM.INITO TO INITO;
  CALC.INITO TO INTEGRAL_TERM.INIT;
  CALC.PREO TO INTEGRAL_TERM.EX;
  INTEGRAL_TERM.EXO TO CALC.POST;
END_CONNECTIONS
DATA_CONNECTIONS
  HOLD TO INTEGRAL_TERM.HOLD;
  PV TO CALC.PV;
  SP TO CALC.SP;
  KP TO CALC.KP;
  KI TO CALC.KI;
  CYCLE TO INTEGRAL_TERM.CYCLE;
  CALC.XOUT TO XOUT;
  CALC.ETERM TO INTEGRAL_TERM.XIN;
  INTEGRAL_TERM.XOUT TO CALC.ITERM;
  0 TO CALC.TD;
  0 TO CALC.DTERM;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
SUBAPPLICATION PI_REAL_APPL (* A Subapplication *)
EVENT_INPUT
 INIT;
 EX;
END_EVENT
EVENT_OUTPUT
 INITO;
 EXO;
END_EVENT
VAR_INPUT
 HOLD: BOOL; (* Hold when TRUE *)
 PV: REAL; (* Process variable *)
 SP: REAL; (* Set point *)
 KP: REAL; (* Proportional gain *)
 KI: REAL; (* Integral gain = Sample period/Reset time *)
 X0: REAL; (* Initial integrator output *)
END_VAR
VAR_OUTPUT XOUT: REAL; END_VAR
FBS
 ETERM: FB_SUB_REAL;
 INTEGRATOR: ACCUM_REAL;
 CALC: PI_CALC;
END_FBS
EVENT_CONNECTIONS
  INIT TO INTEGRATOR.INIT;
  INTEGRATOR.INITO TO INITO;
  EX TO ETERM.REQ;
  ETERM.CNF TO INTEGRATOR.EX;
  INTEGRATOR.EXO TO CALC.EX;
  CALC.EXO TO EXO;
END_CONNECTIONS
DATA_CONNECTIONS
  X0 TO INTEGRATOR.X0;
  HOLD TO INTEGRATOR.HOLD;
  PV TO ETERM.IN1;
  SP TO ETERM.IN2;
  KP TO CALC.KP;
  KI TO CALC.KI;
  ETERM.OUT TO INTEGRATOR.XIN;
  ETERM.OUT TO CALC.ETERM;
  INTEGRATOR.XOUT TO CALC.ITERM;
  CALC.XOUT TO XOUT;
```

```
  1 TO ETERM.QI;
END_CONNECTIONS
END_SUBAPPLICATION
====================================================================
FUNCTION_BLOCK REQUESTER
           (* Service Requester Interface *)
EVENT_INPUT
 INIT WITH QI, PARAMS;    (* Service Initialization *)
 REQ WITH QI, SD_1, SD_m;  (* Service Request *)
END_EVENT
EVENT_OUTPUT
 INITO WITH QO, STATUS;       (* Initialization Confirm *)
 CNF WITH QO, STATUS, RD_1, RD_n; (* Service Confirmation *)
END_EVENT
VAR_INPUT
 QI: BOOL;   (* Event Input Qualifier *)
 PARAMS: ANY; (* Service Parameters *)
 SD_1: ANY;  (* Data to transfer, extensible *)
 SD_m: ANY;  (* Last data item to transfer *)
END_VAR
VAR_OUTPUT
 QO: BOOL;   (* Event Output Qualifier *)
 STATUS: ANY; (* Service Status *)
 RD_1: ANY;  (* Received data, extensible *)
 RD_n: ANY;  (* Last received data item *)
END_VAR
SERVICE REQUESTER/RESOURCE
SEQUENCE normal_establishment
 REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO+();
END_SEQUENCE
SEQUENCE unsuccessful_establishment
 REQUESTER.INIT+(PARAMS) -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE normal_data_transfer
 REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF+(RD_1,...,RD_n);
END_SEQUENCE
SEQUENCE data_transfer_error
 REQUESTER.REQ+(SD_1,...,SD_m) -> REQUESTER.CNF-(STATUS);
END_SEQUENCE
SEQUENCE application_initiated_termination
 REQUESTER.INIT-() -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
SEQUENCE resource_initiated_termination
 -> REQUESTER.INITO-(STATUS);
END_SEQUENCE
END_SERVICE
END_FUNCTION_BLOCK
====================================================================
FUNCTION_BLOCK XBAR_MVCA (* XBAR_MVC + Adapters *)
EVENT_INPUT
 INIT WITH VF,VR,DTL,DT,BKGD,LEN,DIA,DIR; (* Initialize *)
END_EVENT
EVENT_OUTPUT
 INITO;
END_EVENT
VAR_INPUT
 VF: INT:= 20;    (* ADVANCE speed in +%/s *)
 VR: INT:= -40;    (* RETRACT speed in -%/s *)
 DTL: TIME:= t#750ms; (* LOAD Delay *)
 DT: TIME:= t#250ms; (* Simulation Interval *)
 BKGD: COLOR:= COLOR#blue; (* Transfer Bar Color *)
 LEN: UINT:= 5;  (* Bar Length in Diameters *)
 DIA: UINT:= 20; (* Workpiece diameter *)
 DIR: UINT;     (* Orientation: 0=L/R, 1=T/B, 2=R/L, 3=B/T *)
END_VAR
SOCKETS
 LDU_SKT: LD_UNLD;
END_SOCKETS
PLUGS
```

```
 LDU_PLG: LD_UNLD;
END_PLUGS
FBS
 MVC: XBAR_MVC;
END_FBS
EVENT_CONNECTIONS
  INIT TO MVC.INIT;
  MVC.INITO TO INITO;
  MVC.LOADED TO LDU_SKT.UNLD;
  LDU_SKT.LD TO MVC.LOAD;
  MVC.ADVANCED TO LDU_PLG.LD;
  LDU_PLG.UNLD TO MVC.UNLOAD;
  MVC.UNLOADED TO LDU_PLG.CNF;
END_CONNECTIONS
DATA_CONNECTIONS
  LDU_SKT.WO TO MVC.WI;
  LDU_SKT.WKPC TO MVC.LDCOL;
  MVC.WO TO LDU_PLG.WO;
  MVC.WKPC TO LDU_PLG.WKPC;
  VF TO MVC.VF;
  VR TO MVC.VR;
  DTL TO MVC.DTL;
  DT TO MVC.DT;
  BKGD TO MVC.BKGD;
  LEN TO MVC.LEN;
  DIA TO MVC.DIA;
  DIR TO MVC.DIR;
END_CONNECTIONS
END_FUNCTION_BLOCK
====================================================================
```

Annex G
(informative)

Attributes

G.1 General principles

Attributes may be associated with *data types, variables, applications*, and *types* and *instances* of *function blocks, devices, resources*, and their component elements. Attributes have values that may be modified and accessed at various points in the life cycle of the function block type or instance.

In addition to the descriptions of function block *algorithms*, supplementary information is necessary to support the use of a function block during the course of its software life cycle. This information may be provided by attaching *attributes* to the component elements of function block *types* or *instances*.

Attributes can be applied to elements such as *data types, variables*, and *parameters* that are used in the specification of function block types or instances. Graphical language elements may require additional attributes for holding information such as position, color, size, etc.

Attributes can also be applied directly to function block types and instances, for instance to hold the version of a function block type specification.

Certain attributes may be used throughout the life cycle of a function block. For instance, an attribute related to a function block type specification may be accessed when the function block type is selected from a library, when an instance of the function block type is queried, etc.

Other attributes may only exist at certain points in the life cycle. For instance, text defining the purpose of a particular function block instance might be applied only when the function block is instantiated, and might be modified during the life of the function block instance.

Certain function block attributes may be installed in associated *resources* and be accessible during the lifetime of the distributed *application*. Such attributes are typically used to support access to function block parameter values by external devices, e.g., to restrict the values of parameters that may be set using a hand-held configurator to predefined safe limits.

G.2 Attribute definitions

An attribute definition provides the information specified in Table G.1. Each attribute has a name and a data type of its associated value. An attribute may have a default value that will be used until a value is given at some point in the software life cycle. In the example given in G.1, the `DESCRIPTION` attribute has an initial value of '' (the empty string) that may be overwritten with a more meaningful description when a function block instance is configured or even during its active use.

Attributes themselves may require additional information to that shown in Table G.1. Such information is designated as *sub-attributes*.

Table G.1 – Elements of attribute definitions

Element	**Example**	
Name	`DESCRIPTION`	
Data type	`WSTRING(30)`	
Default value	`""`	
Associated element	**Function block types**	**Function block instances**
Usage	Configuration	Run-time

G.3 Examples

NOTE The following examples are for the purpose of illustrating the use of attributes and are not to be considered as normative definitions of standard attributes.

An example of a *data type attribute* is:

- `Max_System_Value` - This attribute defines the maximum supported value of a numeric data type. It is applied to the generic data type `ANY_NUM`, so that all numeric types such as `INT` and `REAL` will inherit this attribute. Note that each specific data type will have its own value for this attribute, and that standard values for this attribute for some data types are given in Table E.1.

Examples of attributes that apply to *variables* are:

- `Diagnostic_Access` – This determines whether the value of a variable is accessible by a run-time diagnostic system.
- `Write_Access` – This defines the access level required to change the value of a variable, e.g., `'Operator', 'System', 'Diagnostics'`.
- `Units` – The dimensional units that apply to a variable, e.g., `'l', 'm/s', 'cm'`.
- `Usage` – A multi-line textual description of the usage of the associated variable.

Examples of function block type attributes are:

- `Usage_Class` – This describes the general usage of the function block, e.g., `'Input', 'Output', 'Control'`.
- `Version` – This describes the version number of the function block type definition, e.g., `'1.2'`.
- `Help` – A multi-line textual description that may be accessed at various points in the life cycle.

Attributes which are relevant to the scheduling of *algorithms* for *execution* include:

- `ExecutionTime` – This attribute, of type `TIME`, specifies the worst-case time for execution of a particular *algorithm* of a specified *function block type* in a particular *resource type*.
- `Priority` – This attribute is associated with a particular *event connection* within a *resource*, and may be inherited from the *resource type*. This attribute may be used by a resource which supports pre-emptive *multitasking* to determine the priority of *execution* of an *algorithm* invoked by an *EC action* associated with an *EC state* which is activated by an event with the specified priority.

G.4 Attribute sources

Attributes may come from the following main sources:

- **Implicit** attributes such as function block *type names*, *instance names*, *variable names* and their *data types*, are defined as part of the normal *declaration* process for the function block.
- **Standard** attributes are those which are required as part of a standard, such function block type versions, maximum range of parameters, parameter descriptions, etc.
- **Product-specific** attributes are those which a system vendor has provided, such as function block type product codes, hardware addresses of function block instances, etc.
- **Application-specific** attributes are those which a system developer specifies to support the use of a particular data type or function block in an application, such as an additional function block instance identifier to fit a customer's desired style, a fail-safe default value for output parameters, an alternative parameter description in a national language, etc.

G.5 Attribute inheritance

Function block elements will inherit attributes from more primitive elements. For instance, a *variable* within a *function block type declaration* will inherit attributes of its associated *data type*, and a function block *instance* will inherit attributes of the associated function block *type*.

Data types will inherit attributes down the generic type hierarchy defined in IEC 61131-3. For example, attributes applied to `ANY_REAL` will also apply to `LREAL` and `REAL`.

G.6 Declaration syntax

The assignment of an attribute value to a declared element is similar to assigning a value to an *instance* of an *attribute type* in which the instance has the same name as the type.

The declaration of an attribute *type* uses the same syntax as the declaration of a *data type* as defined in IEC 61131-3, with the exception that the delimiting keywords are `ATTRIBUTE...END_ATTRIBUTE` instead of `TYPE...END_TYPE`. For instance, the declaration of the attribute type `DESCRIPTION` in Table G.1 would be:

```
ATTRIBUTE DESCRIPTION: WSTRING(30); END_ATTRIBUTE
```

The assignment of a value to an attribute *instance* uses the same syntax as that for assigning an initial value to a *variable* as described in IEC 61131-3, with the following extensions:

a) the name of the attribute instance is the same as the name of the corresponding attribute type;
b) no data type is specified for the attribute instance;
c) the value assignment is enclosed in the **pragma** construct defined in IEC 61131-3;
d) multiple attribute value assignments, separated by semicolons, may be included in the pragma construct;
e) the pragma construct shall be located in such a manner that the declaration to which it applies can be determined unambiguously.

An example of the application of these rules is:

```
FUNCTION_BLOCK PID
     {DESCRIPTION:= "Proportional + Integral + Derivative Control;
      AUTHOR:= "JHC"; VERSION:= "19990103/JHC"}
     INPUT_EVENT
      INIT WITH QI, PARAMS; {DESCRIPTION:= "Initialization Request"}
     ...etc.
```

Bibliography

IEC 60050-351:2006, *International Electrotechnical Vocabulary – Part 351: Control technology*

IEC 61131-5:2000, *Programmable controllers – Part 5: Communications*

IEC 61499 (all parts), *Function blocks*

IEC 61499-2:2012, *Function blocks – Part 2: Software tools requirements*

IEC 61499-4, *Function blocks – Part 4: Rules for compliance profiles*

ISO/IEC 7498-4, *Information processing systems – Open systems interconnection – Basic reference model – Part 4: Management framework*

ISO/IEC 8825-1:2008, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

ISO/IEC 10040:1998, *Information technology – Open Systems Interconnection – Systems management overview*

ISO/IEC/IEEE 60559, *Information technology – Microprocessor systems – Floating-point arithmetic*

ISO 2382 (all parts), *Information technology – Vocabulary*

This page deliberately left blank

British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards -based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at bsigroup.com/standards or contacting our Customer Services team or Knowledge Centre.

Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at bsigroup.com/shop, where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to bsigroup.com/subscriptions.

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member**.

PLUS is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit bsigroup.com/shop.

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email bsmusales@bsigroup.com.

Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

Copyright

All the data, software and documentation set out in all British Standards and other BSI publications are the property of and copyrighted by BSI, or some person or entity that owns copyright in the information used (such as the international standardization bodies) and has formally licensed such information to BSI for commercial publication and use. Except as permitted under the Copyright, Designs and Patents Act 1988 no extract may be reproduced, stored in a retrieval system or transmitted in any form or by any means – electronic, photocopying, recording or otherwise – without prior written permission from BSI. Details and advice can be obtained from the Copyright & Licensing Department.

Useful Contacts:

Customer Services
Tel: +44 845 086 9001
Email (orders): orders@bsigroup.com
Email (enquiries): cservices@bsigroup.com

Subscriptions
Tel: +44 845 086 9001
Email: subscriptions@bsigroup.com

Knowledge Centre
Tel: +44 20 8996 7004
Email: knowledgecentre@bsigroup.com

Copyright & Licensing
Tel: +44 20 8996 7070
Email: copyright@bsigroup.com

BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK

bsi.

...making excellence a habit.™