IEC-PAS 61499-1

Edition 1.0
2000-09

Function blocks for industrial-process measurement and control systems

Part 1: Architecture

PUBLICLY AVAILABLE SPECIFICATION

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

Reference number
IEC/PAS 61499-1

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

FUNCTION BLOCKS FOR INDUSTRIAL-PROCESS MEASUREMENT AND CONTROL SYSTEMS –

Part 1: Architecture

FOREWORD

A PAS is a technical specification not fulfilling the requirements for a standard, but made available to the public and established in an organization operating under given procedures.

IEC-PAS 61499-1 has been processed by IEC technical committee 65: Industrial-process measurement and control.

The text of this PAS is based on the following document:	This PAS was approved for publication by the P-members of the committee concerned as indicated in the following document:
Draft PAS	**Report on voting**
65/248/PAS	65/252/RVD

Following publication of this PAS, the technical committee or subcommittee concerned will investigate the possibility of transforming the PAS into an International Standard.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. GENERAL REQUIREMENTS

1.1. Scope

This Specification 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 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*.

NOTE - This Specification 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. Clause 5 of this Part addresses the extent to which the elements defined in this Specification may be restricted by the functional capabilities of compliant systems, subsystems, and devices.

Part of the purpose of this specification 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 Specification 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 Specification be concise, implementable, complete, unambiguous, and consistent.

NOTE 1 The provisions of this Specification alone are not sufficient to ensure interoperability among devices of different vendors. Standards complying with this Specification may specify additional provisions to ensure such interoperability.

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

This Specification consists of two Parts:

- Part 1, "Architecture", contains:
  - general requirements, including an introduction, 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 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;
  - requirements to be met by compliant systems and standards.
- Part 2, "Engineering task support", will present guidance for the support of engineering tasks in the design, implementation, operation and maintenance of distributed industrial-process measurement and control systems constructed according to the architecture defined in this Part.

1.2. Normative references

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

IEC 600050-351(1998?), International Electrotechnical Vocabulary Chapter 351: Automatic Control (2nd.Ed.)

IEC 559 (1989), Binary floating-point arithmetic for microprocessors

IEC 617-12 (1983), Graphical symbols for diagrams, Part 12: Binary logic elements

IEC 65B/373/CD, Committee Draft - IEC 61131-3, Programmable controllers, Part 3: Programming languages, 2nd Ed., 1998-11-27.

ISO 2382 (various Parts and dates), Information processing systems - Vocabulary

ISO 8601:1988, Data elements and interchange formats - Information interchange - Reresentations of dates and times

ISO/AFNOR, Dictionary of Computer Science, 1989, ISBN 2-12-4869111-6

ISO/IEC 7498-1, Information Technology - Open Systems Interconnection - Basic Reference Model, 1994

ISO/IEC 8824: 1990, Information technology - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)

ISO/IEC 8825: 1990, Information technology - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)

ISO TR 8509-1987, Information processing systems - Open Systems Interconnection - Service conventions

ISO/IEC 10040-1992, Information technology - Open Systems Interconnection - Systems management overview

ISO/IEC 10646-1:1993, Information technology - Universal multiple-octet coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane

1.3. Definitions

NOTE 1 - Terms defined in this clause are *italicized* where they appear in the bodies of definitions.

NOTE 2 - The ISO/AFNOR *Dictionary of computer science* and the *International Electrotechnical Vocabulary* should be consulted for terms not defined or referenced in this specification.

1.3.1. Definitions from other standards

NOTE Definitions are written out in this document for convenience. To avoid duplication, the terms alone will be listed in the final International Standard.

For the purposes of this specification, the following terms as defined in IEC 60050-351 apply:

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

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

Notes: 1 -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)
2 - 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.

For the purposes of this specification, the following terms as defined in the various Parts of ISO 2382 apply:

NOTE - Definition numbers from ISO 2382 are given in parentheses following the definition.

data type: A set of values together with a set of permitted *operations*. (15.04.01)

data: A reinterpretable representation of *information* in a formalized manner suitable for communication, interpretation or processing. (01.01.02)

functional unit: An *entity* of *hardware* or *software,* or both, capable of accomplishing a specified purpose. (01.01.40)

mapping: A set of values having defined correspondence with the quantities or values of another set. (02.04.05)

message: An ordered series of *characters* intended to convey *information*. (16.02.01)

message sink: That part of a communication *system* in which *messages* are considered to be received. (16.02.03)

message source: That part of a communication *system* from which *messages* are considered to originate. (16.02.02)

network: An arrangement of *nodes* and interconnecting *branches*. (01.01.44)

operation: A well-defined action that, when applied to any permissible combination of known *entities*, produces a new *entity*. (02.10.01)

parameter: A *variable* that is given a constant value for a specified *application* and that may denote the application. (02.02.04)

For the purposes of this specification, the following terms as defined in the ISO/AFNOR *Dictionary of Computer Science* apply:

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

connection: An association established between *functional units* for conveying *information*.

hardware: Physical equipment, as opposed to programs, procedures, rules and associated documentation.

information: The meaning that is currently assigned to *data* by means of the conventions applied to that data.

For the purposes of this specification, the following term as defined in the document *IEC DIS 61508-4: Functional safety - Safety-related systems - Part 4: Definitions and Abbreviations of Terms* applies:

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

1.3.2. Additional definitions

The following terms are defined for the purposes of this specification.

1.3.2.1. **acceptor:** A *function block instance* which provides a *socket adapter* of a defined *adapter interface type*.

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

1.3.2.3. **adapter connection:** A *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.

1.3.2.4. **adapter interface type:** A *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*.

1.3.2.5. **algorithm:** A finite set of well-defined rules for the solution of a problem in a finite number of *operations*.

1.3.2.6. **application:** A *software functional unit* that is specific to the solution of a problem in industrial-process measurement and control.

NOTE: An application may be distributed among *resources*, and may communicate with other applications.

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

1.3.2.8. **basic function block type:** a *function block type* which cannot be decomposed into other function blocks and which utilizes an *execution control chart (ECC)* to control the *execution* of its *algorithms*.

1.3.2.9. **bidirectional transaction:** A *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

1.3.2.10. **communication connection:** A *connection* which utilizes the "communication mapping function" of one or more *resources* for the conveyance of *information*.

1.3.2.11. **communication function block:** A *service interface function block* which represents the *interface* between an *application* and the "communication mapping function" of a *resource*.

1.3.2.12. **communication function block type:** A *function block type* whose *instances* are *communication function blocks*.

1.3.2.13. **component function block:** A *function block instance* which is used in the specification of an *algorithm* of a *composite function block type.*

NOTE - A component function block can be of *basic, composite* or *service interface type*.

1.3.2.14. **component subapplication:** A *subapplication instance* which is used in the specification of a *subapplication type.*

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

1.3.2.16. **concurrent:** Pertaining to *algorithms* that are *executed* during a common period of time during which they may have to alternately share common *resources*.

1.3.2.17. **configuration (of a *system* or *device*) :** A step in system design: selecting *functional units*, assigning their locations and defining their interconnections.

1.3.2.18. **configuration (of a *programmable controller system*) :** A language element corresponding to a *programmable controller system* as defined in IEC 61131-1.

1.3.2.19. **configuration parameter:** A *parameter* related to the *configuration* of a *system, device* or *resource.*

1.3.2.20. **confirm primitive:** A *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*.

1.3.2.21. **critical region:** An *operation* or a 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.

1.3.2.22. **data connection:** An association between two *function blocks* for the conveyance of *data*.

1.3.2.23. **data input:** An *interface* of a *function block* which receives *data* from a *data connection.*

1.3.2.24. **data output:** An *interface* of a *function block* which supplies *data* to a *data connection.*

1.3.2.25 **declaration:** The mechanism for establishing the definition of an *entity*. A declaration may involve attaching an *identifier* to the entity, and allocating *attributes* such as *data types* and *algorithms* to it.

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

NOTE - A *programmable controller system* as defined in IEC 61131-1 is a *device* in the terms of this Specification.

1.3.2.27. **device management application:** An *application* whose primary function is the management of a multiple *resources* within a *device.*

1.3.2.28. **entity:** A particular thing, such as a person, place, *process*, object, concept, association, or *event*.

1.3.2.29. **event:** An instantaneous occurrence that is significant to scheduling the *execution* of an *algorithm*.

NOTE - The execution of an algorithm may make use of *variables* associated with an event.

1.3.2.30. **event connection:** An association among *function blocks* for the conveyance of *events*.

1.3.2.31. **event input:** An *interface* of a *function block* which receives *events* from an *event connection.*

1.3.2.32. **event input variable (EI variable):** A Boolean *variable* corresponding to an *event input.*

1.3.2.33. **event output:** An *interface* of a *function block* which issues *events* to an *event connection.*

1.3.2.34. **event output variable (EO variable):** A Boolean *variable* corresponding to an *event output.*

1.3.2.35. **exception:** An *event* that causes suspension of normal *execution*.

1.3.2.36. **execution:** The process of carrying out a sequence of *operations* specified by an *algorithm.*

NOTE - 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.

1.3.2.37. **execution control action (EC action):** An element associated with an *execution control state* which identifies an *algorithm* to be *executed* and an *event* to be issued on completion of execution of the algorithm.

1.3.2.38. **execution control chart (ECC):** A 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*.

1.3.2.39. **execution control initial state (EC initial state):** The *execution control state* which is active upon initialization of an *execution control chart*.

1.3.2.40. **execution control state (EC state):** A situation in which the behavior of a *basic function block* with respect to its *variables* is determined by the *algorithms* associated with the *execution control state* through its *execution control action*.

1.3.2.41. **execution control transition (EC transition):** The condition whereby control passes from a predecessor *execution control state* to a successor *execution control state*.

1.3.2.42. **function:** A specific purpose of an *entity* or its characteristic action.

1.3.2.43. **function block (function block instance):** A *software functional unit* comprising an individual, named copy of a data structure and associated *operations* specified by a corresponding *function block type*.

NOTE 1 - Typical operations of a function block include modification of the values of the data in its associated data structure.

NOTE 2 - 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.

1.3.2.44. function block network: A *network* whose nodes are *function blocks* or *subapplications* and their *parameters* and whose branches are *data connections* and *event connections*.

NOTE - This is a generalization of the *function block diagram* defined in IEC 61131-3.

1.3.2.45. identifier: One or more *characters* used to name an *entity*.

1.3.2.46. implementation: The development phase in which the *hardware* and *software* of a *system* become operational.

1.3.2.47. indication primitive: A *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*.

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

NOTE - An *input parameter* of a *function block*, as defined in IEC 61131-3, is an *input variable.*

1.3.2.49. instance: A *functional unit* comprising an individual, named *entity* with the *attributes* of a defined *type.*

1.3.2.50. instance name: An *identifier* associated with and designating an *instance.*

1.3.2.51. instantiation: The creation of an *instance* of a specified *type.*

1.3.2.52. internal operations (of a *function block*): *Operations* associated with an *algorithm* of a function block, with its *execution* control, or with the functional capabilities of the associated *resource*.

1.3.2.53. internal variable: A *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*.

1.3.2.54. invocation: The process of initiating the *execution* of the sequence of *operations* specified in an *algorithm.*

1.3.2.55. literal: A lexical unit that directly represents a value.

1.3.2.56. management function block: A *function block* whose primary *function* is the management of *applications* within a *resource.*

1.3.2.57. management resource: A *resource* whose primary *function* is the management of other *resources.*

1.3.2.58. model: A representation of a real world process, *device*, or concept.

1.3.2.59 multitasking: A mode of operation that provides for the *concurrent execution* of two or more *algorithms*.

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

NOTE - An *output parameter* of a *function block*, as defined in IEC 61131-3, is an *output variable.*

1.3.2.61. plug adapter: An *instance* of an *adapter interface type* which provides a starting point for an *adapter connection* from a *provider* function block.

1.3.2.62. provider: A *function block instance* which provides a *plug adapter* of a defined *adapter interface type*.

1.3.2.63. request primitive: A *service primitive* which represents an interaction in which an *application* invokes some *algorithm* provided by a *service*.

1.3.2.64. requester: A *functional unit* which initiates a *transaction* via a *request primitive.*

1.3.2.65. **resource:** A *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- The RESOURCE defined in IEC 61131-3 is a programming language element corresponding to the *resource* defined above.

NOTE 2 - A *device* contains one or more resources.

1.3.2.66. **resource management application:** An *application* whose primary function is the management of a single *resource.*

1.3.2.67. **responder:** A *functional unit* which concludes a *transaction* via a *response primitive.*

1.3.2.68. **response primitive:** A *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*.

1.3.2.69. **sample:** sense and retain the instantaneous value of a *variable* for later use.

1.3.2.70. **scheduling function:** A *function* which selects *algorithms* or *operations* for *execution,* and initiates and terminates such execution.

1.3.2.71. **service:** A functional capability of a *resource* which can be modeled by a sequence of *service primitives*.

1.3.2.72. **service interface function block:** A *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*.

1.3.2.73. **service primitive:** An abstract, implementation-independent representation of an interaction between an *application* and a *resource*.

1.3.2.74. **socket adapter:** An *instance* of an *adapter interface type* which provides an end point for an *adapter connection* to an *acceptor* function block.

1.3.2.75. **software:** Intellectual creation comprising the programs, procedures, rules, *configurations* and any associated documentation pertaining to the operation of a *system*.

1.3.2.76. **software tool:** *Software* that is used for the production, inspection or analysis of other software.

1.3.2.77. **subapplication instance:** An *instantiation* of a *subapplication type* inside an *application* or inside a subapplication type.

NOTE - 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.

1.3.2.78. **subapplication type:** A *functional unit* which allows the creation of substructures of *applications* in the form of a fractal hierarchy. The body of a subapplication type consists of interconnected *component function blocks* or *component subapplications*.

1.3.2.79. **transaction:** A 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.

1.3.2.80. **type:** A *software* element which specifies the common *attributes* shared by all *instances* of the type.

1.3.2.81. **type name:** An *identifier* associated with and designating a *type.*

1.3.2.82. **unidirectional transaction:** A *transaction* in which a request and possibly *data* is conveyed from an *requester* to a *responder*, and in which a response is not conveyed from the responder back to the requester.

1.3.2.83. **variable:** A *software entity* that may take different values, one at a time.

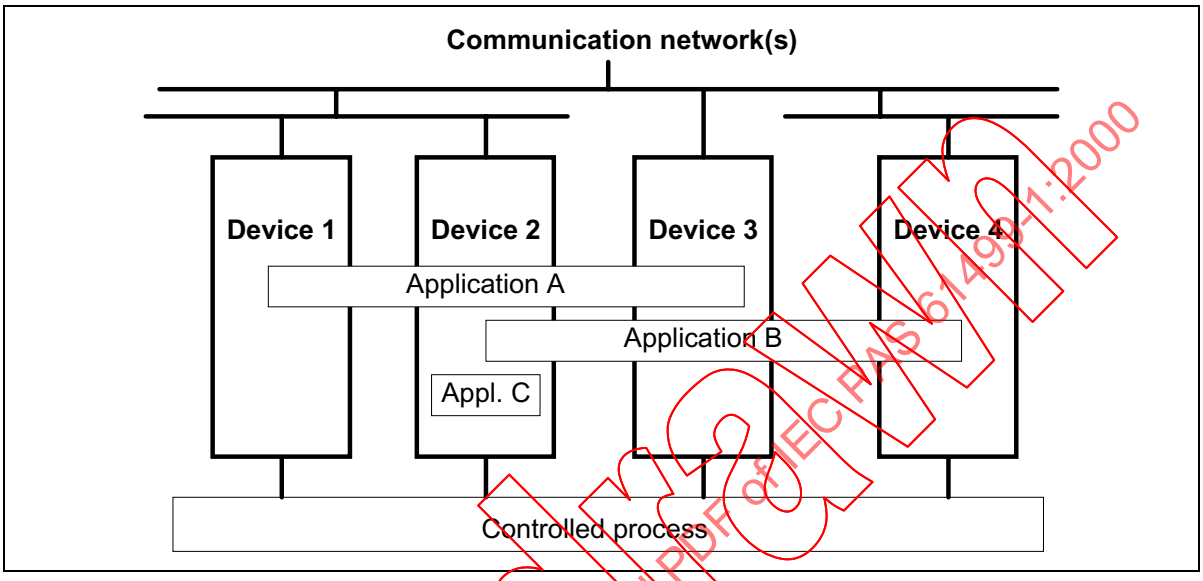
NOTE 1 - The values of a variable are usually restricted to a certain *data type.*

NOTE 2 - Variables may be classified as *input variables, output variables,* and *internal variables.*

1.4. Reference models

1.4.1. System model

For the purposes of this specification, an industrial process measurement and control *system* (IPMCS) is modeled, as shown in figure 1.4.1, as a collection of *devices* interconnected and communicating with each other by means of one or more communication *networks*. These networks may be organized in a hierarchical manner.



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

Figure 1.4.1 - 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 1.4.1, or may be distributed among several devices, such as applications A and B in figure 1.4.1. 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.

1.4.2. Device model

As illustrated in figure 1.4.2, a *device* shall contain at least one *interface,* that is, process interface or communication interface, and can contain zero or more *resources* and *function block networks*..

NOTE 1 A device is considered to be an *instance* of a corresponding device *type*, defined as specified in clause 4 of this Part.

NOTE 2 A device that contains no resources is considered to be functionally equivalent to a *resource* as defined in 1.4.3.

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.

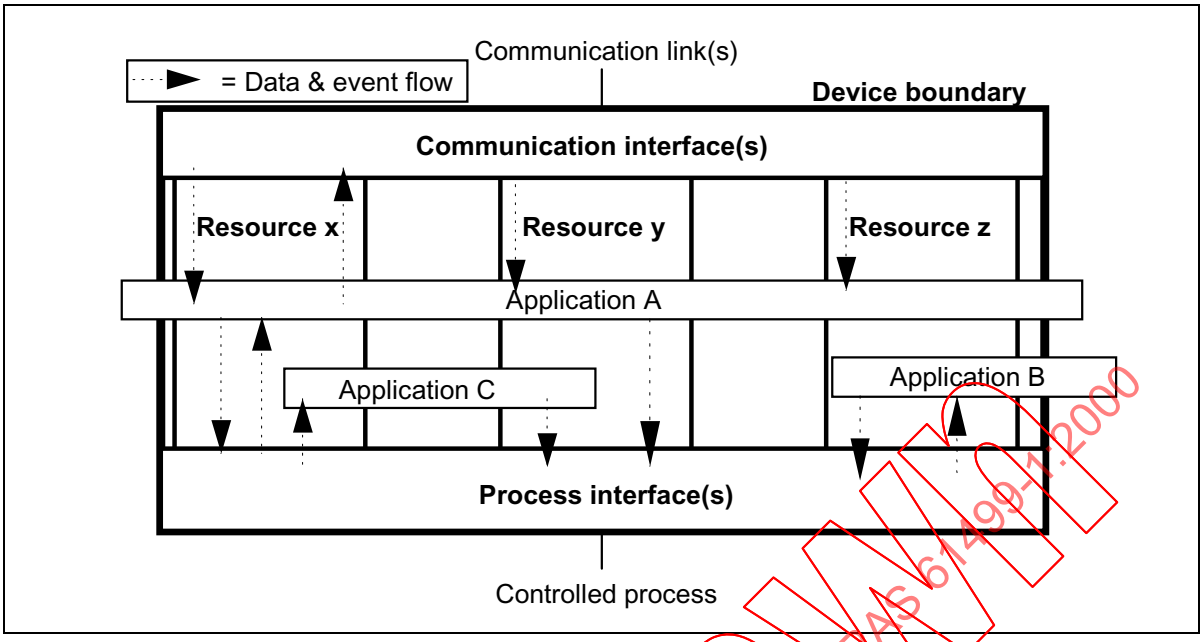


Figure 1.4.2 - Device model
(example: Device 2 from figure 1.4.1)

1.4.3. Resource model

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

NOTE 1 - A resource is considered to be an *instance* of a corresponding resource *type*, defined as specified in clause 4 of this Part.

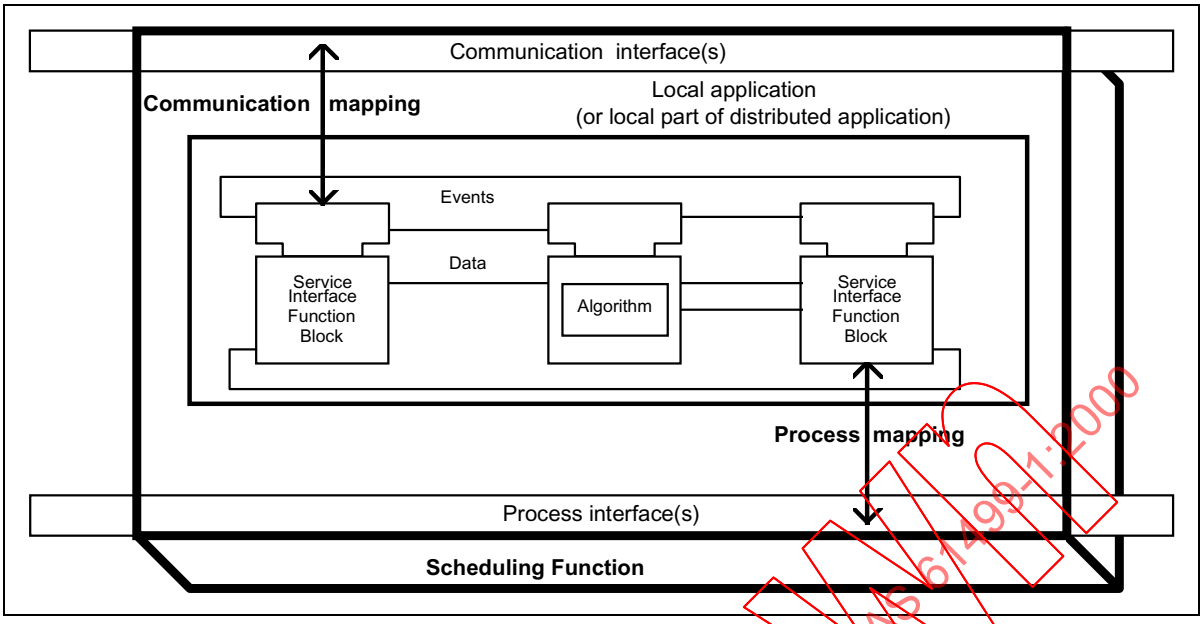
NOTE 2 - Although a resource has independent control of its operation, its operational states may need to be coordinated with those of other resources for the purposes of installation, test, etc.

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 - The consideration of other possible aspects of resources is beyond the scope of this Specification.

As illustrated in figure 1.4.3, 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 1.4.3, 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 1.4.3, 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: 1) the occurrence of events; 2) function block interconnections; and 3) scheduling information such as periods and priorities. Means of achieving traditional scheduling functions such as cyclic execution of a *function block network* are described in IEC 61499-3.



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 may be shared among resources.

Figure 1.4.3 - Resource model

1.4.4. Application model

For the purposes of this specification, 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*.

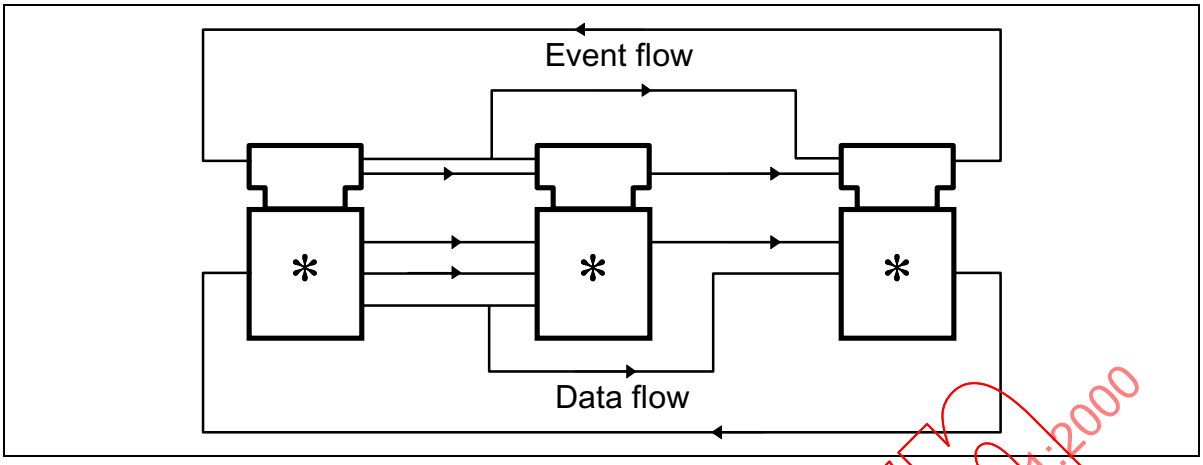
Subapplications are *instances* of *subapplication types*, which like applications consist of *function block networks*. Application names, subapplication and function block *instance names* may therefore be used to create a hierarchy of *identifiers* that can uniquely identify every *function block 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 specification, applications are defined by *function block networks* specifying event and data flow among *function block* or *subapplication instances,* as illustrated in figure 1.4.4. 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 clause 2 of this Part.

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



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

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

Figure 1.4.4 - Application model

1.4.5. Function block model

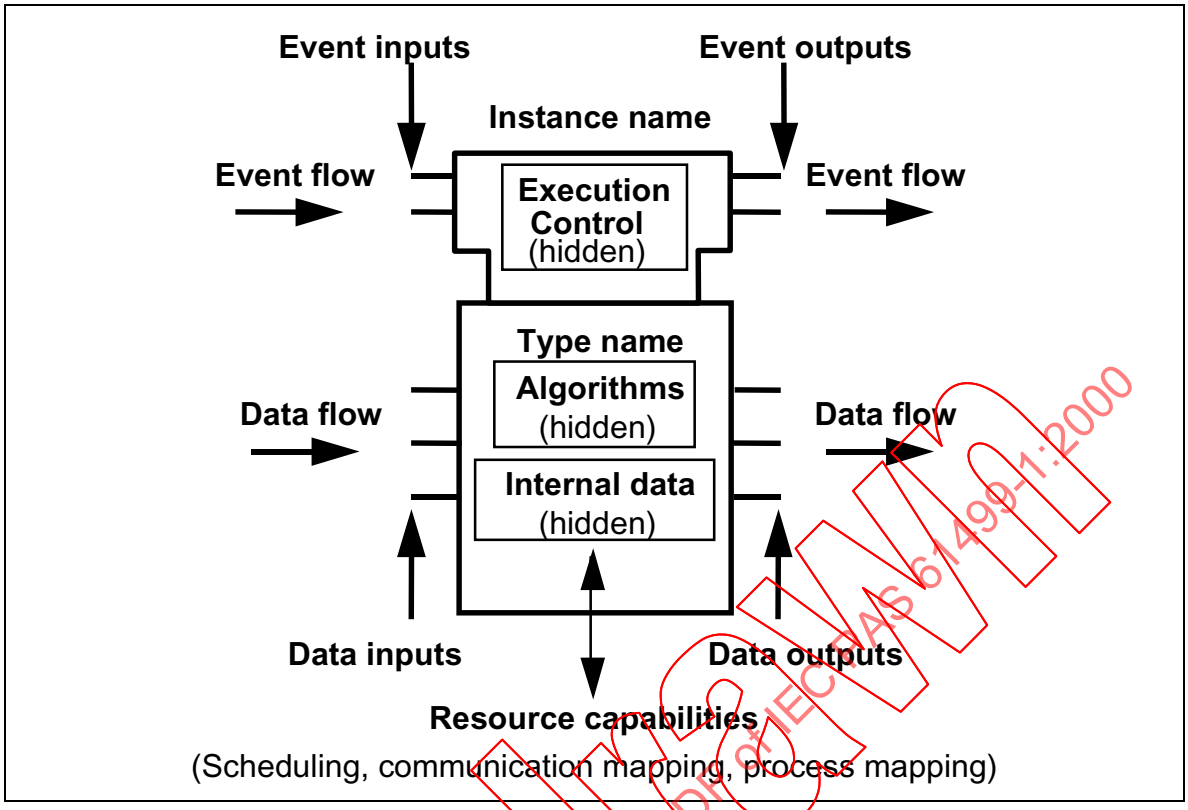
A *function block (function block instance)* is a *functional unit* of *software* comprising an individual, named copy of the data structure specified by a *function block type*, which persists from one *invocation* of the function block to the next. The characteristics of function block instances are described in subclause 1.4.5.1, and function block type specifications are described in subclause 1.4.5.2.

1.4.5.1. Characteristics of function block instances

A *function block instance* exhibits the following characteristic features as illustrated in figure 1.4.5.1:

- 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 may 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 1.4.5.1 - 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.

NOTE - Access to internal variables and state information of function block instances may be provided by additional functional capabilities of the associated resource, as illustrated in 3.3.

Means for specifying the causal relationships among event inputs, event outputs, and execution of algorithms are defined in clauses 2 and 3 of this Part.

1.4.5.2. 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 2.2.1.

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 2.2.1.2 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 2.2.1.3.
- The functionality of an *instance* of a *composite function block type* or a *subapplication type* is declared, using the mechanisms defined in 2.3.1 and 2.4.1 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 3.1.

1.4.5.3. Execution model for basic function blocks

As shown in figure 1.4.5.3-1, 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.

NOTE 1 - *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,
- the "communication mapping", "process mapping", "scheduling", or other functional capability of the *resource.*

NOTE 2 - Execution control in *composite function blocks* is achieved via event flow within the function block body.

Figure 1.4.5.3-1 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 2.2.1.1) 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 2.2.1.1.

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.

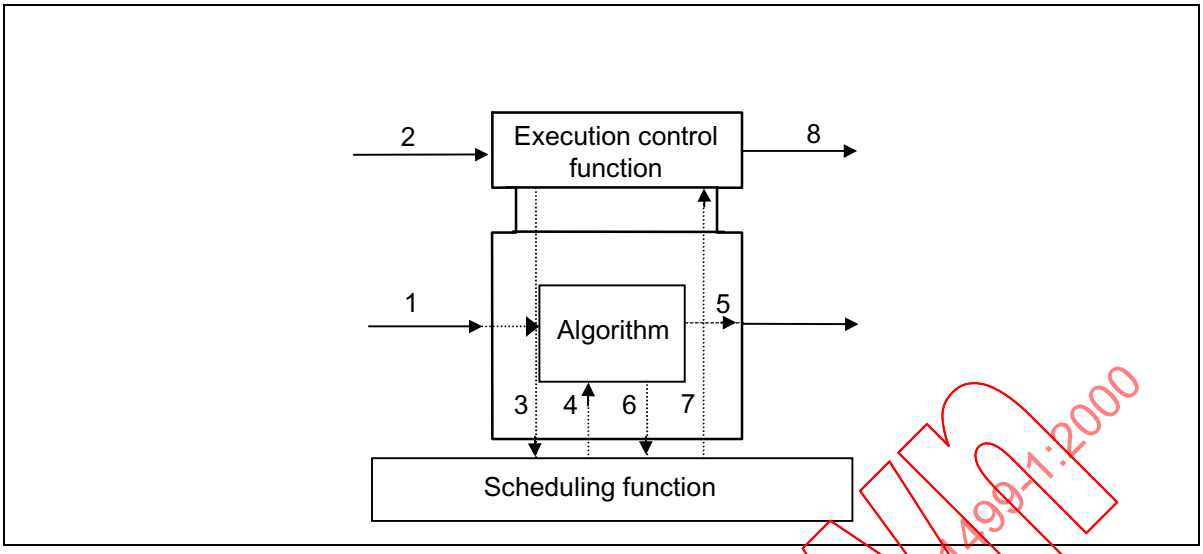
As shown in figure 1.4.5.3-2, the significant timing delays in this case which are of interest in application design are:

$T_{setup} = t_2 - t_1$

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

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

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



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

Figure 1.4.5.3-1 - Execution model

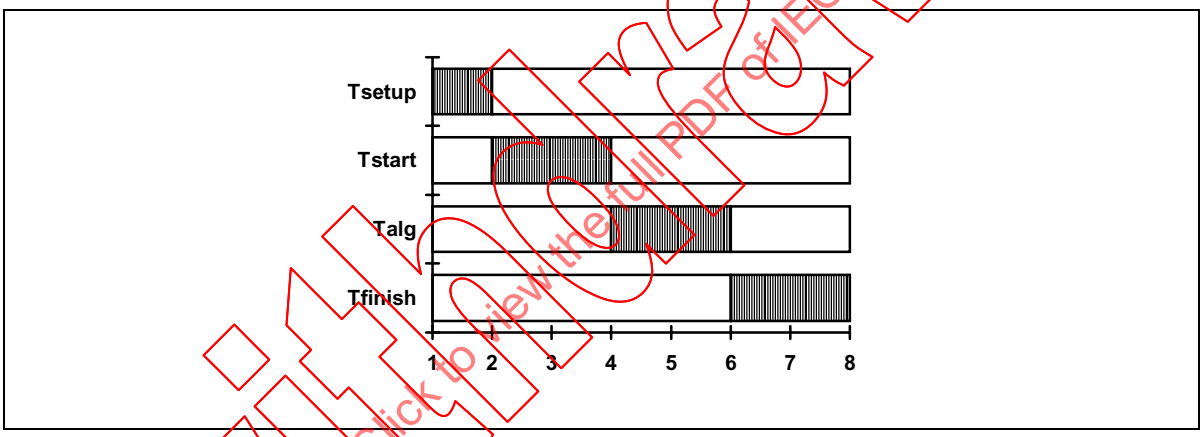


Figure 1.4.5.3-2 - Execution timing

NOTE - The axis labels 1,2,... in the above figure correspond to the times t1, t2,... in the preceding text.

Normative requirements for the specification of function block execution control in the general case (which includes the above case) are defined in clause 2 of this Part.

NOTE 1 - Depending on the problem to be solved, various requirements may exist for the synchronization of the values of *input variables* with the *execution* of *algorithms*. Such requirements may 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.

Users of this Specification should be aware that the results of algorithm execution may be unpredictable if such requirements are not met.

ANNEX I of this Part describes mechanisms to meet the above requirements.

NOTE 2 - *Resources* may need to schedule the *execution* of *algorithms* in a *multitasking* manner. The specification of attributes to facilitate such scheduling is described in ANNEX J.

1.4.6. Distribution model

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 must form an atomic unit of distribution. That is, all the elements contained in a given function block instance must 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 of this Part apply when applications or subapplications are distributed among multiple resources:

- Clause 4 of this Part defines the requirements for the case where multiple applications or subapplications are distributed among multiple resources and devices.
- Clause 3 of this Part defines the requirements for communication services to support distribution of applications or subapplications among multiple devices.

1.4.7. Management model

Figure 1.4.7 provides a schematic representation of the management of *resources* and *devices*. Figure 1.4.7(a) illustrates a case in which a *management resource* provides shared facilities for management of other *resources* within a device, while figure 1.4.7(b) 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 - Subclause 3.3 defines *service interface function block types* for management of *applications,* and Annex G provides examples of their usage.

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

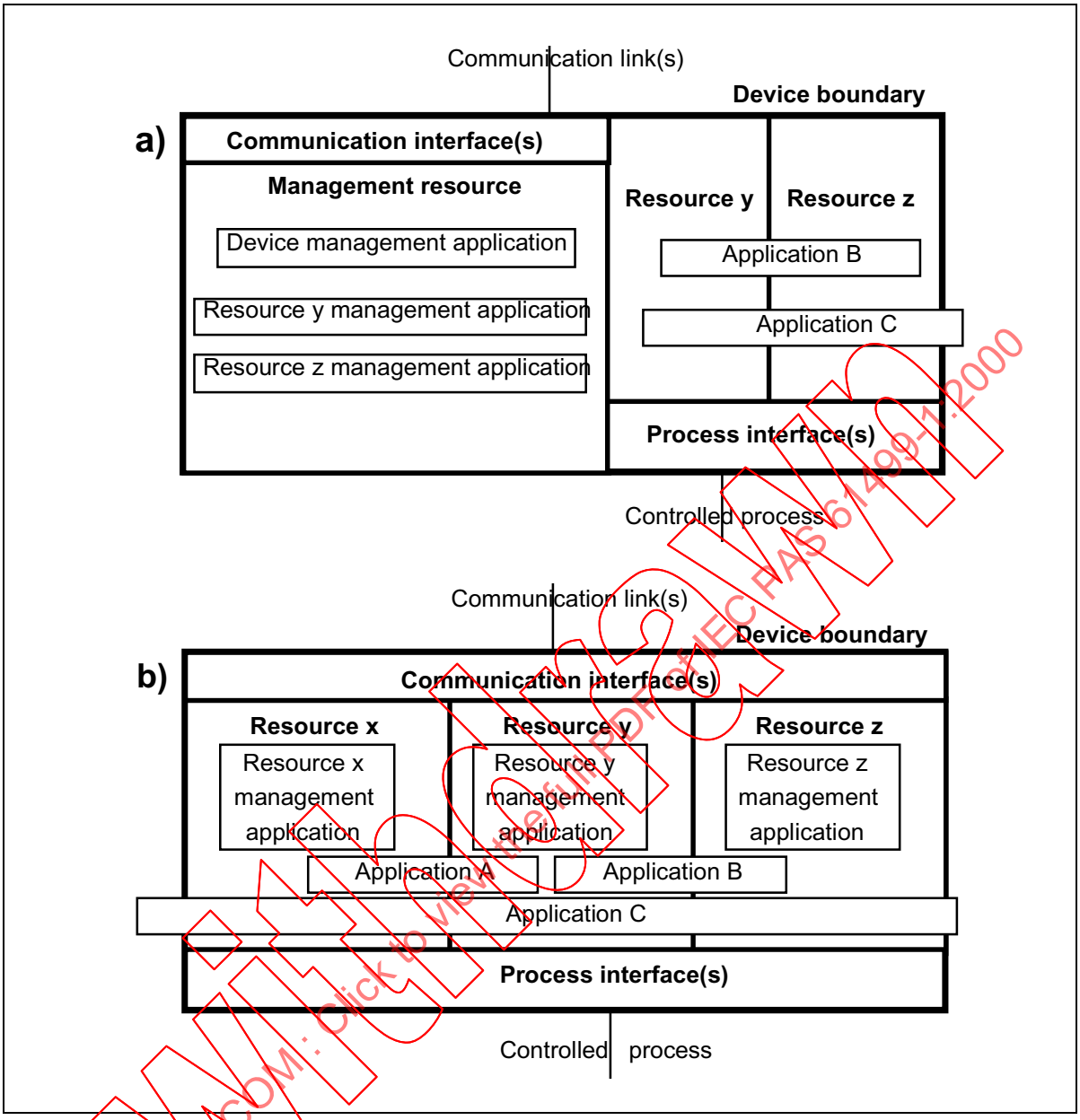


Figure 1.4.7 - Management models -a) Shared, b) Distributed

1.4.8. 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" must 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 subclause 3.3.3.

2. FUNCTION BLOCK AND SUBAPPLICATION TYPE SPECIFICATION

2.1. Overview

As illustrated in figure 2.1, this clause defines the means for the type specification of three kinds of blocks:

- Subclause 2.2 defines the means for specifying and determining the behavior of instances of *basic function block types,* as illustrated in figure 2.1(a). 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 clause 5 of this Part.
- Subclause 2.3 defines the means for specifying *composite function block types,* as illustrated in figure 2.1(b). 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 2.4 defines the means for specifying *subapplication types,* as illustrated in Figure 2.1(c). 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.*

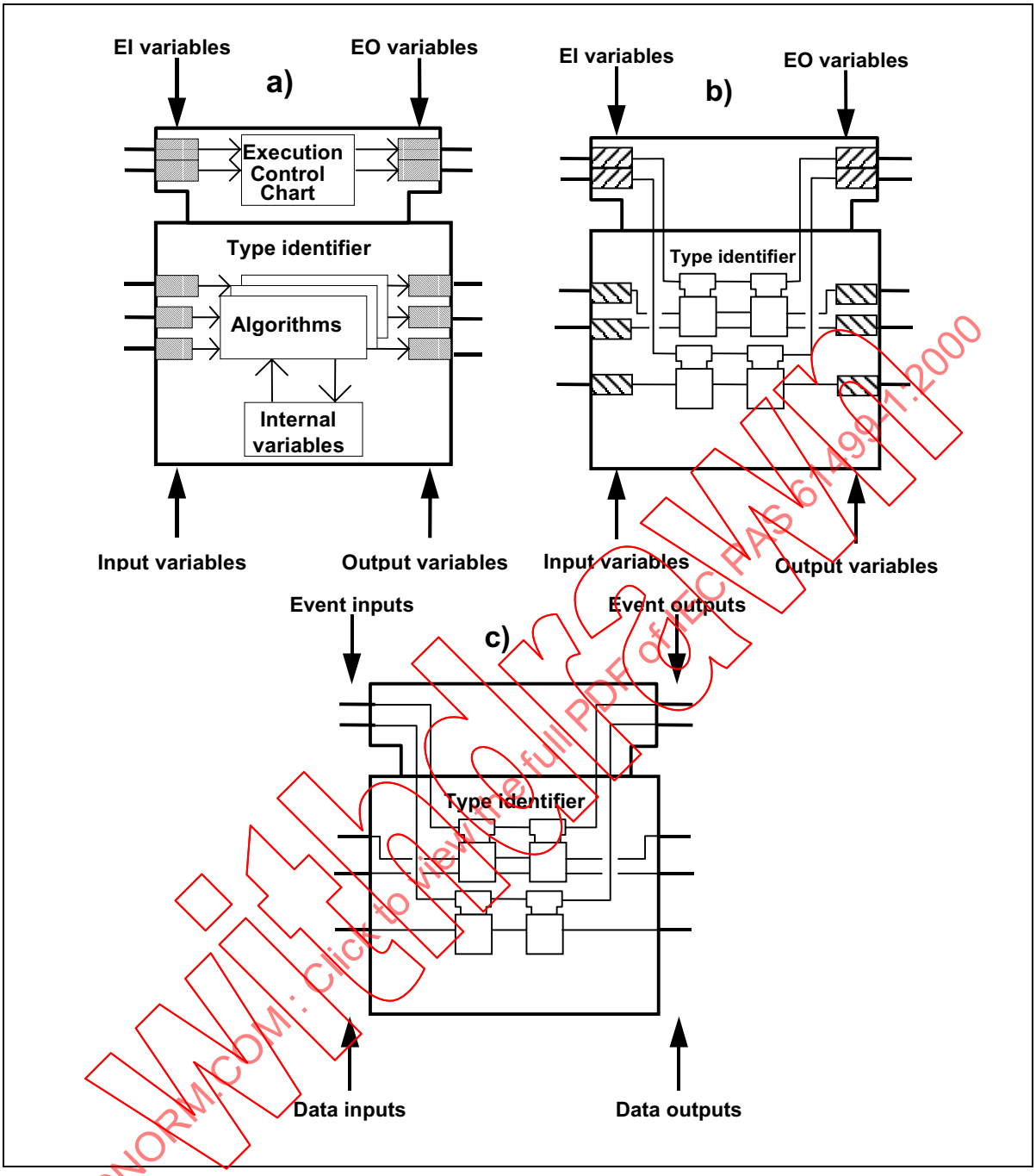


Figure 2.1 - Function block and subapplication types
a) Basic function block (subclause 2.2), b) Composite function block (subclause 2.3), c) Subapplication (subclause 2.4)

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

2.2. Basic function blocks

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

2.2.1. Type declaration

As illustrated in figure 2.2.1, a *basic function block type* can be declared textually according to the syntax specified in Annex B.2 or graphically according to the following rules:

1. The function block *type name* shall be shown at the top center of the main rectangular block.
2. The *input* and *output variable* names and *type declarations* shall be shown at the left and right edges of the main rectangular block, respectively.
3. The *interface* of the function block type to *events* shall be declared as specified in subclause 2.2.1.1 of this Part.
4. The *algorithms* associated with the function block type shall be declared as specified in subclause 2.2.1.2 of this Part.
5. Control of the *execution* of the associated algorithms shall be declared as specified in subclause 2.2.1.3 of this Part.

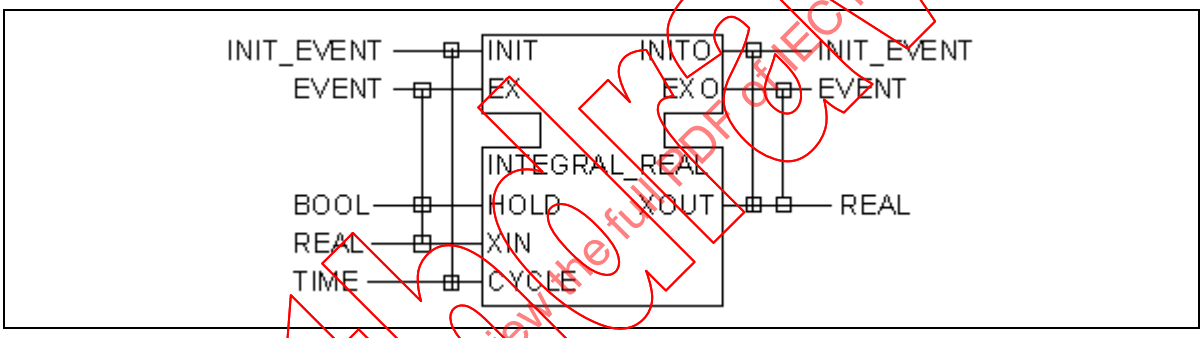


Figure 2.2.1 - Basic function block type declaration

NOTE 1 - See Annex H for a textual declaration of this example.
NOTE 2 - This example is illustrative only. Details of the specification are not normative.

2.2.1.1. Event interface declaration

As shown in figure 2.2.1, the *interface* of a *basic function block type* to *events* can be declared textually according to the syntax given in Annex B.2, or graphically according to the following rules:

1. The event interface shall have the form of the common control block outline specified as No. 12-05-02 of IEC 617-12.
2. *Event input* names shall be shown at the left-hand side of the control block.
3. *Event output* names shall be shown at the right-hand side of the control block.
4. Event *types* shall be 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 2.2.1 and Annex H, the `WITH` qualifier or a graphical equivalent can be used to specify which *input variables* or *output variables* shall be *sampled* upon the occurrence of an *event* at the associated *event input* or *event output*, respectively, as described in 2.2.2.2. This information may be used to determine the required communication *services* when *configuring* a distributed *application* as described in clause 4 of this Part. Each *input variable* and *output variable* shall appear in at least one `WITH` clause or its graphical equivalent.

NOTE 5 -It is a consequence of the above requirement that provision is made for the *sampling* of all input variables.

NOTE 6 - See 1.4.5.3 for an application of the WITH qualifier to the execution model of a basic function block.

NOTE 7 - See Annex C.3 of IEC 61499-2 for examples of the use of the WITH qualifier to determine communication service requirements.

2.2.1.2. Algorithm declaration

As shown in Annex H, *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 Part.

2.2.1.3. 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 2.2.1.2 (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 shall be represented and interpreted as follows:

1. The ECC shall be included in an *execution control block* section of the function block type declaration, encapsulated by the control block construct.
2. The ECC shall contain exactly one *EC initial state,* represented graphically as a round or rectangular, double-outlined shape with an associated *identifier*. The EC initial state shall have no associated EC actions.
3. The ECC shall contain one or more *EC states,* represented graphically as round or rectangular, single-outlined shapes, each with an associated *identifier*.
4. The ECC can utilize (but not modify) *event input (EI) variables* having the same names as the event inputs of the control block. The associations of these variables to the event inputs shall be as defined in 2.2.2.
5. The ECC can utilize and/or modify *event output (EO) variables* having the same names as the event outputs of the control block. The associations of these variables to the event outputs shall be as defined in 2.2.2.
6. The ECC can utilize but not modify variables declared in the function block type specification.
7. An *EC state* can have zero or more associated *EC actions.* The association of the EC actions with the EC state shall be expressed in graphical or textual form.
8. 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.
9. An *EC transition* shall be represented graphically or textually as a directed link from one EC state to another (or to the same state).
10. Each EC transition shall have an associated Boolean condition, equivalent to a Boolean expression utilizing one or more *event input variables, input variables, output variables,* or *internal variables* of the function block.

Figure 2.2.1.3 illustrates the elements of an ECC. An equivalent textual declaration using the syntax of Annex B.2 is given in Annex H.

NOTE 1 According to the model given in subclause 2.2.2, the evaluation of an EC transition condition is disabled until the algorithms associated with its predecessor EC state have completed their execution. Therefore, for example, the `1` in the EC transitions following the EC states `INIT` and `MAIN` is equivalent in this case to the use of `INIT0` and `EX0`, respectively.

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.

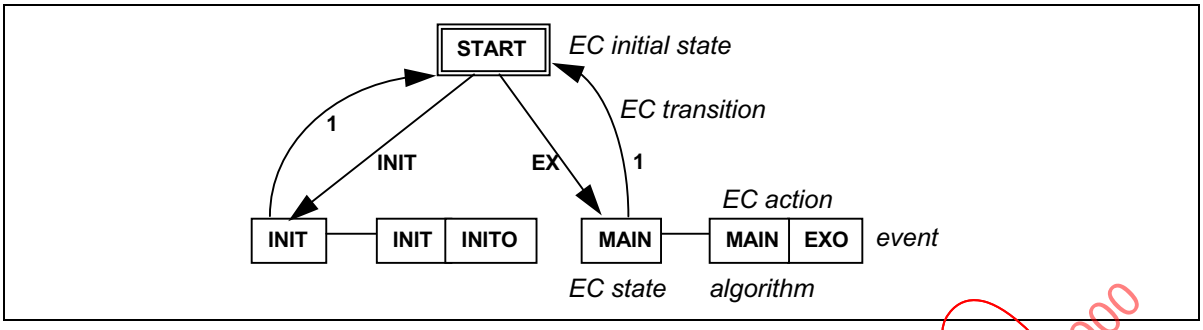


Figure 2.2.1.3 - ECC example

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,. e.g., 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.

2.2.2. Behavior of instances

2.2.2.1. Initialization

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

1. 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.
2. The values of all *EI variables* and *EO variables* shall be reset to `FALSE (0)`.
3. 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.
4. The *EC initial state* of the function block's *Execution Control Chart (ECC)* shall be activated and all other *EC states* shall be deactivated.

NOTE - The conditions under which a resource shall perform 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 2.2.1.3. 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 2.2.1.

2.2.2.2. 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 may 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 be according to the following rules:

1. The resource shall maintain for each event input an *EI variable* plus a storage element which exhibits the behavior defined by the state machine in figure 2.2.2.2-1 and table 2.2.2.2-1.
2. Operation of the ECC shall exhibit the behavior defined by the state machine in figure 2.2.2.2-2 and table 2.2.2.2-2.
3. The evaluation of an *EC transition* condition is disabled until the *algorithms* associated with its predecessor *EC state* have completed their *execution*.

NOTE 2 It is a consequence of this model that multiple occurrences of an event at the same event input may be lost at transition t2 in figure 2.2.2.2-1. The detection and processing of such loss (for example as an *exception* or a *fault* as described in subclause 2.6) is an **implementation-dependent** feature.

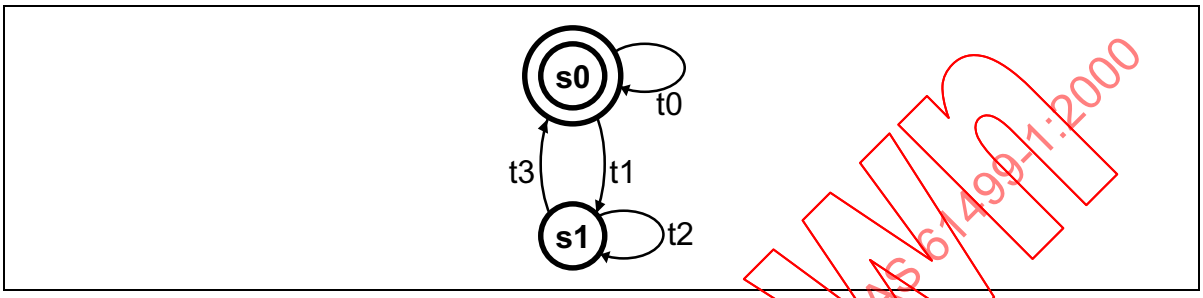


Figure 2.2.2.2-1 - Event input state machine

Table 2.2.2.2-1 - States and transitions of event input state machine

state	**condition**	
s0	waiting for event	
s1	waiting for ECC to finish	
transition	**condition**	**operation**
t0	input mapped [a]	none
t1	event arrives	ECC invocation request [b]
t2	event arrives	**implementation-dependent**
t3	input mapped [a]	none

[a] This confirmation is issued by the ECC state machine shown in Figure 2.2.2.2-2.
[b] This operation consists of requesting the resource to invoke the ECC.

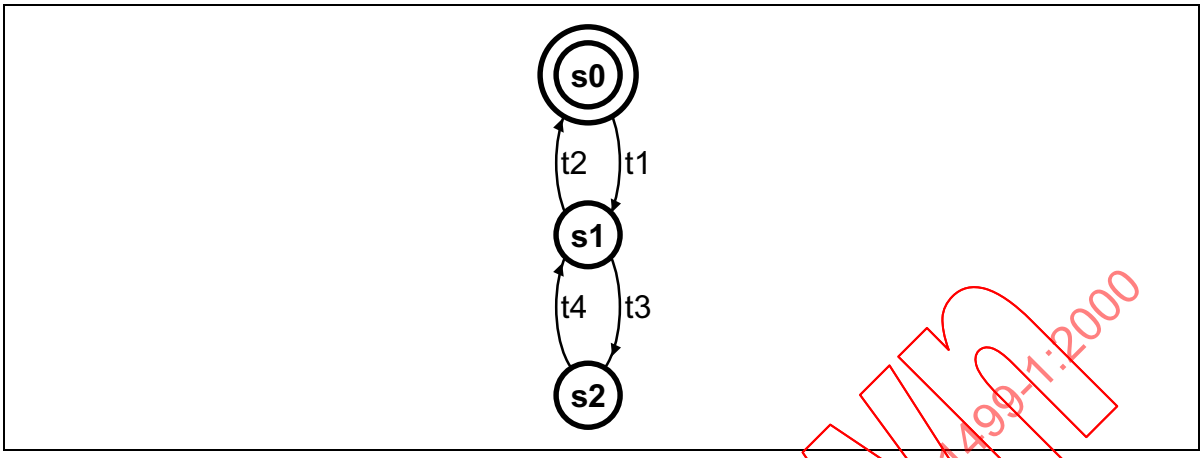


Figure 2.2.2.2-2 - ECC operation state machine

Table 2.2.2.2-2 - States and transitions of ECC operation state machine

state	condition	
s0	idle	
s1	scheduling algorithms	
s2	waiting for algorithms to complete	
transition	**condition**	**operations**
t1	invoke ECC [a]	set EI variables [b] confirm input mapping [c] evaluate transitions [d] (NOTE)
t2	no transition clears	issue events [e]
t3	a transition clears	schedule algorithms [f]
t4	algorithms complete [g]	clear EI variables [h] set EO variables [i] evaluate transitions [d] (NOTE)

NOTE - *Software tools* may provide means for determining the order in which the EC transitions following an active EC state are to be evaluated.

[a] This condition is issued by the *resource* at an **implementation-dependent** time after it has received one or more "ECC invocation requests" from one or more of the event input state machines specified in Figure 2.2.2.2-1 and Table 2.2.2.2-1.

[b] This operation consists of setting the values of the corresponding EI input variables to TRUE (1) and *sampling* the input variables associated with the event inputs by WITH declarations as described in 2.2.1.1. For each event and its associated set of input variables, this sampling shall be implemented as a *critical region*.

[c] This operation consists of issuing "input mapped" confirmations to all of the event input state machines of the function block.

[d] This operation consists of evaluating the conditions at all the EC transitions following the active EC state and clearing the first EC transition (if any) for which a TRUE condition is found. "Clearing the EC transition" consists of deactivating its predecessor EC state and activating its successor EC state.

[e] This operation consists, for each event output for which the value of its associated *EO variable* is TRUE (1), of *sampling* each *output variable* associated with the event output by a WITH declaration as described in 2.2.1.1, then issuing an event at the event output followed by resetting the value of the associated EO variable to FALSE (0). This sampling shall be implemented as a *critical region*.

[f] This operation consists of requesting the resource to schedule for execution the algorithms named in the EC actions associated with the active EC state.

[g] This condition consists of the completion of execution of all the algorithms associated with the active EC state (always TRUE for an EC state which has zero associated algorithms).

[h] This operation consists of resetting to FALSE (0) the values of all the EI variables used in evaluating the transition conditions of the previously cleared transition.

[i] This operation consists of setting to TRUE (1) the value of the EO variables named in the EC action blocks of the active EC state.

2.2.2.3. 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 2.2.2.2.

NOTE The minimum necessary condition for expected response of a *resource* is that the maximum execution time for all algorithms resulting from the arrival of an event at an output of a *service interface function block* must be less than one-half the minimum inter-event arrival time of all such events. Additional constraints may be required depending on the characteristics of the *application* and *resource*.

2.3. Composite function blocks

2.3.1. Type specification

The declaration of *composite function block types* shall follow the rules given in subclause 2.2.1 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:

1. Each event input of the composite function block shall be 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 of Annex A may be employed.

2. Each event input of a component function block shall be 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 of Annex A may be employed.

3. Each event output of a component function block shall be 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 of Annex A may be employed.
4. Each event output of the composite function block shall be 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 of Annex A may be employed.
5. Use of the `WITH` qualifier in the declaration of event inputs and event outputs of composite function block types is permitted. However, it is not required that each input variable and output variable be associated with at least one event input or output respectively as in the case of *basic* or *service interface* function blocks. Use of the `WITH` qualifier results in the sampling of the associated data inputs and outputs as in the case of basic or interface service function blocks.

   NOTE 1 Software tools may provide means of elimination of redundant sampling in the implementation phase.
6. *Instances* of *subapplication types* as defined in subclause 2.4 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:

1. 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.
2. 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.
3. 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.
4. 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 2 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.

Figure 2.3.1-1 illustrates the application of these rules to the example `PI_REAL` function block. Figure 2.3.1-1(a) shows the graphical representation of the external interfaces and 2.3.1-1(b) shows the graphical construction of its body. Figure 2.3.1-2 shows the interfaces and execution control for the function block type `PID_CALC` used in the body of the `PI_REAL` example.

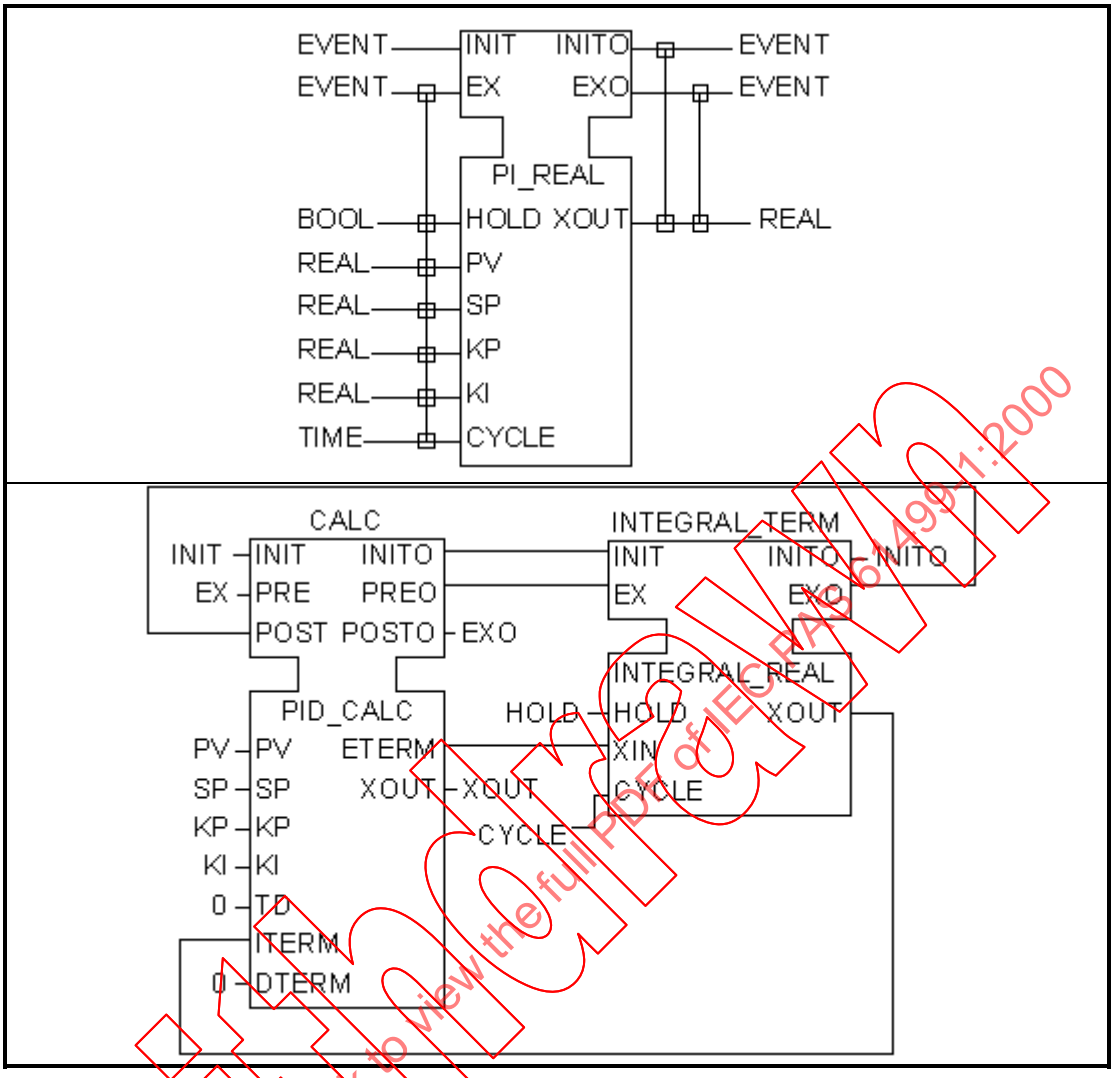
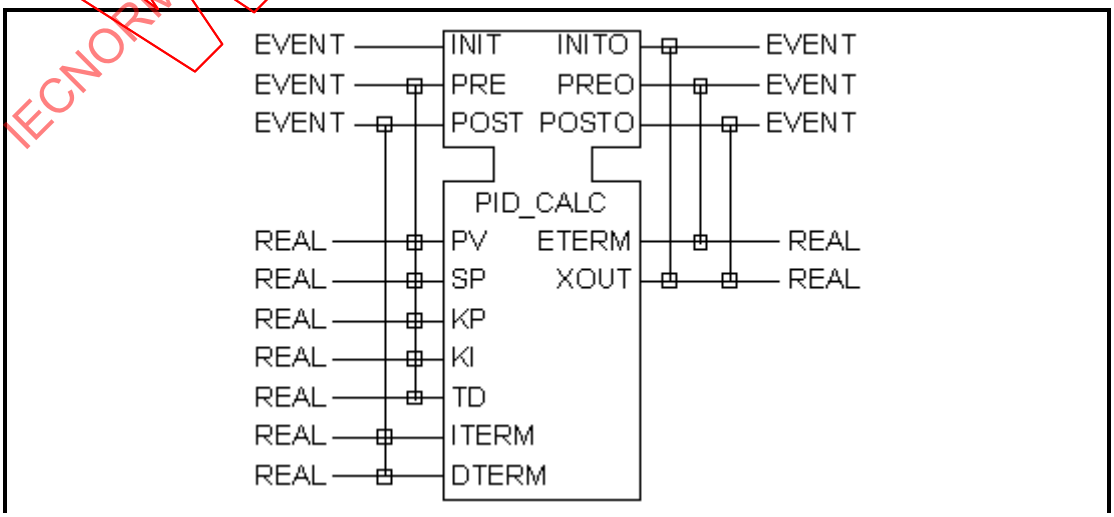


Figure 2.3.1.1 - Composite function block `PI_REAL` example
a) External interfaces, b) Graphical body

NOTE 3 A full textual declaration of this function block type is given in Annex H.

NOTE 4 This example is illustrative only. Details of the specification are not normative.



b)

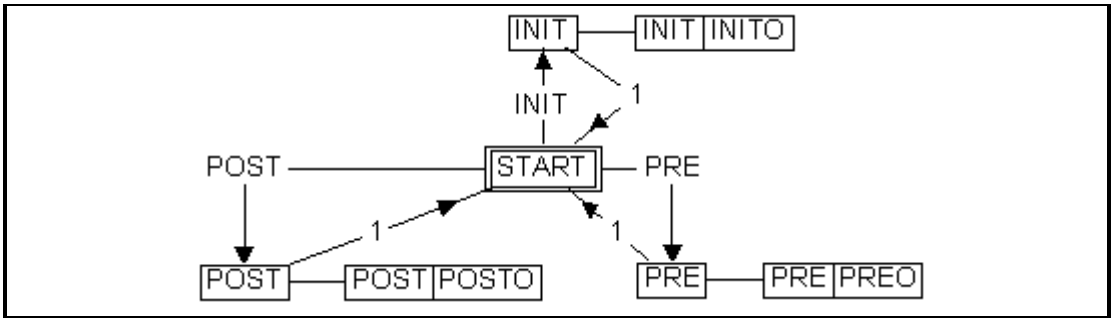


Figure 2.3.1-2 - Basic function block `PID_CALC` example
a) External interfaces, b) Execution control

NOTE 5 A full textual declaration of this function block type is given in Annex H.

NOTE 6 This example is illustrative only. Details of the specification are not normative.

2.3.2. Behavior of instances

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

1. 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.
2. 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.
3. 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.
4. 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 subclause 2.2.2.1.

2.4. Subapplications

2.4.1. Type specification

The declaration of *subapplication types* is similar to the declaration of *composite function block types* as defined in subclause 2.2.1, with the exception that the delimiting keywords shall be `SUBAPPLICATION..END_SUBAPPLICATION`. The following rules shall apply to this usage:

1. The `WITH` qualifier shall not be used in the declaration of event inputs and event outputs of *subapplication types*, since *sampling* of inputs and outputs is only provided for *function block instances* and not for *subapplication instances*.
2. 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.

3. Each event input of a component function block or component subapplication shall be 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.

4. Each event output of a component function block or component subapplication shall be 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.

5. Each event output of the subapplication shall be 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 may 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:

1. 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.

2. 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.

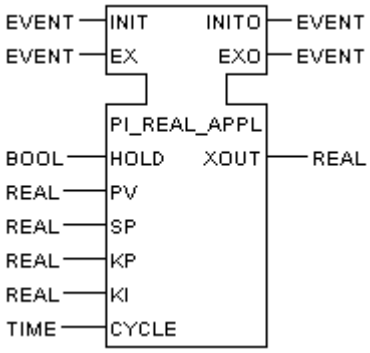
3. Each data output of a component function block or component subapplication can be connected to zero or more data inputs of zero or more component function blocks or component subapplications, or to zero or more data outputs of the subapplication, or both.

4. 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.

EXAMPLE - Figure 2.4.1 illustrates the application of these rules to the example PI_REAL_APPL subapplication. Figure 2.4.1(a) shows the graphical representation of its external interfaces and 2.4.1(b) 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 2.3.1, which is shown in Figure 2.3.1-2.

a)



b)

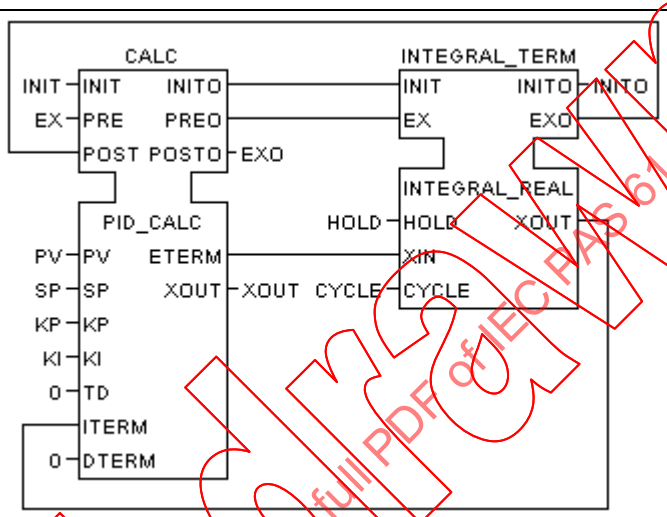


Figure 2.4.1 - Subapplication PI_REAL_APPL example
a) External interfaces, b) Graphical body

NOTE 3 A full textual declaration of this subapplication type is given in Annex H.

NOTE 4 This example is illustrative only. Details of the specification are not normative.

2.4.2. Behavior of instances

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

1. 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.
2. 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.
3. 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.

4. 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.

2.5. Adapter interfaces

Adapter interfaces can be used to provide a compact representation of a specified set of event and data flows. As illustrated in figure 2.5, 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 subclause 3.1.2.

NOTE A given *function block type* may 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*.

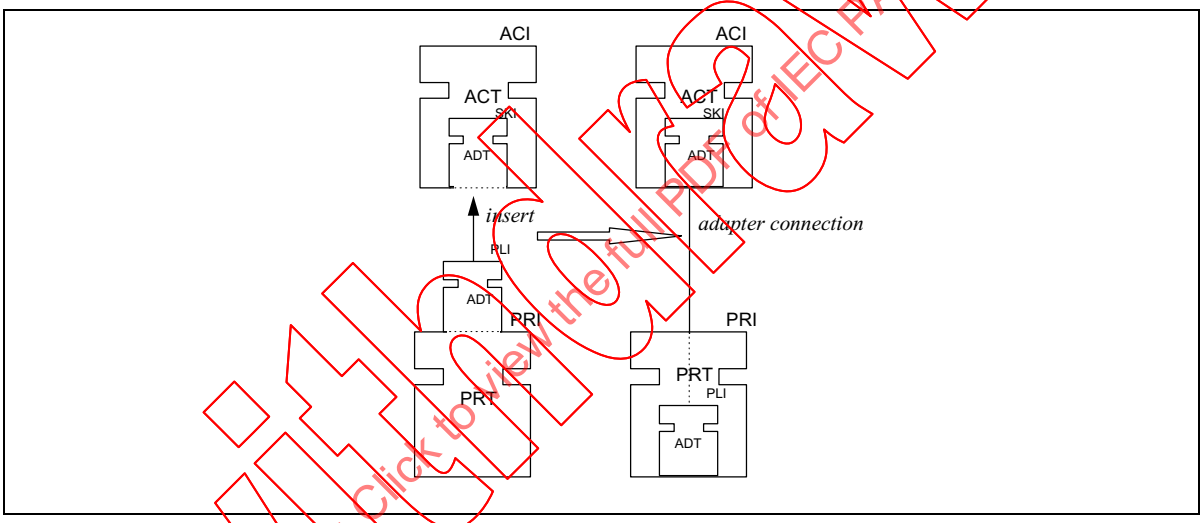


Figure 2.5 - Adapter interfaces - Conceptual model
PRT - Provider type, PRI - Provider instance
ACT - Acceptor type, ACI - Acceptor instance
ADT - Adapter type, PLI - Plug instance, SKI - Socket instance

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

2.5.1. Type specification

An *adapter interface type declaration* shall define only the *interface type* name and its contained *event* and *data interfaces.* These shall be 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 at the beginning of subclause 2.2.1 and subclause 2.2.1.1, 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 Annex B.7.

EXAMPLE - The adapter interface illustrated in Figure 2.5.1 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 2.5.1(b), the typical operation of this interaction consists of the following sequence:

1. 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.
2. 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.
3. 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.

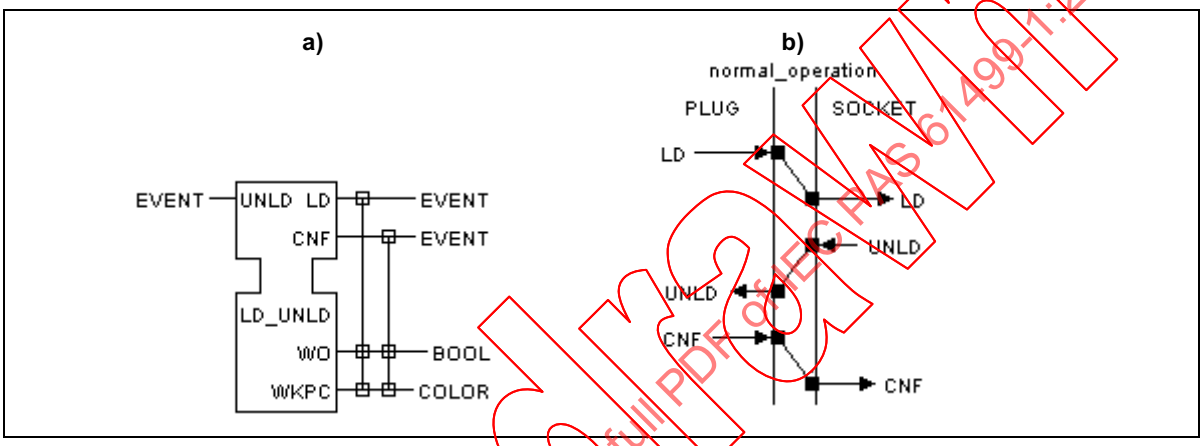


Figure 2.5.1 - Adapter type declaration - graphical example
a) Interface, b) Normal operation (see 3.1.2)

NOTE 1 - A full textual declaration of this adapter type is given in Annex H.
NOTE 2 - This example is illustrative only. Details of the specification are not normative.

2.5.2. Usage

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

1. 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 shall be shown as *output variables* with specialized textual or graphical indication that they are not ordinary output variables.
2. 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 shall be shown as *input variables* with specialized textual or graphical indication that they are not ordinary input variables.
3. *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.
4. *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.
5. Insertion of *plugs* into *sockets* shall be specified in an `ADAPTER_CONNECTIONS ... END_CONNECTIONS` block in the *declaration* of the *resource type, resource instance,* or *composite function block type* containing the respective *provider* and *acceptor* instances.

6. In the body of a *composite function block type,* a *socket* is represented as a *function block* with the same pinout as the corresponding *adapter interface type*. Similarly, a *plug* is represented as a function block with the inputs and outputs of the corresponding adapter interface type reversed.
7. Insertion of plugs into sockets shall be subject to the following constraints:
   - A plug can only be inserted into a socket of the same *adapter interface type.*
   - A plug can only be inserted into zero or one socket at a time.
   - A socket can only accept zero or one plug at a time.
   - A plug can only be inserted in a socket if both are in the same *resource.*
8. *Management function blocks* as described in 3.3 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 2.5.2-1 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.

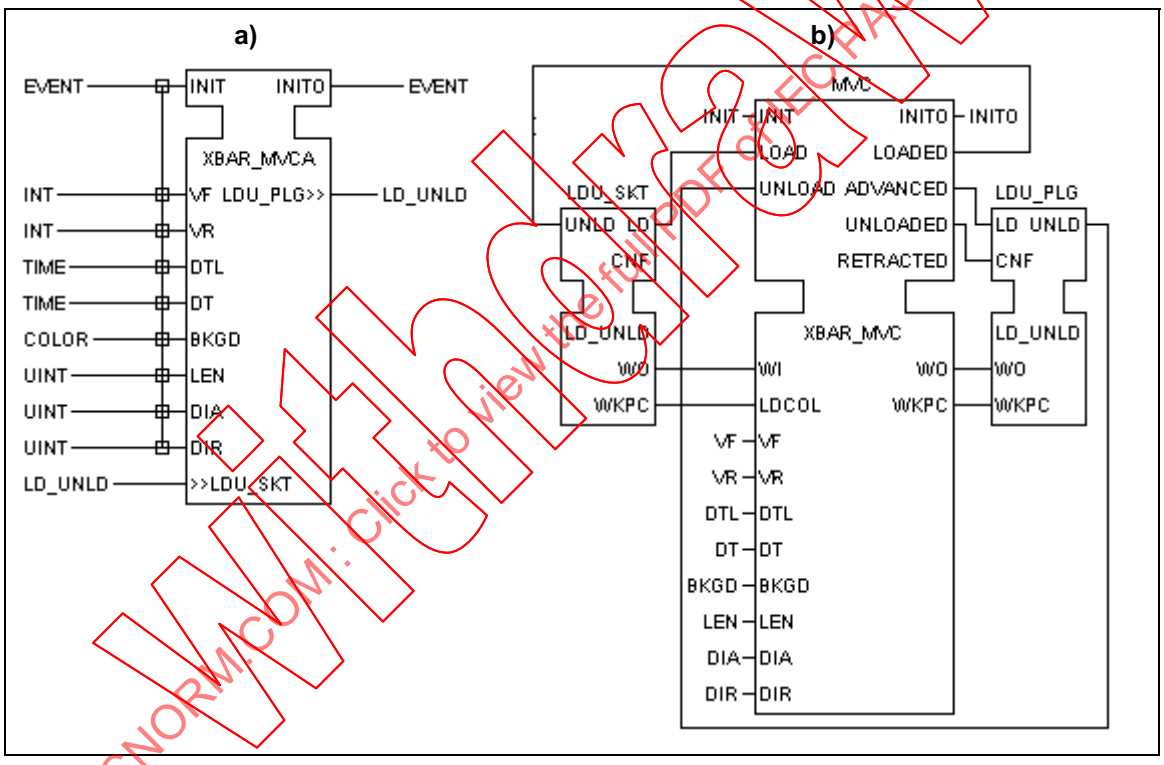


Figure 2.5.2-1 - Illustration of provider and acceptor function block type declarations a) Interface, b) Body

NOTE 1 - A full textual declaration of this example is given in Annex H.
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* may be either *basic* or *composite*.

EXAMPLE 2 - Figure 2.5.2-2 illustrates a *resource configuration* containing two instances of the XBAR_MVCA type illustrated in Figure 2.5.2-1. 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.

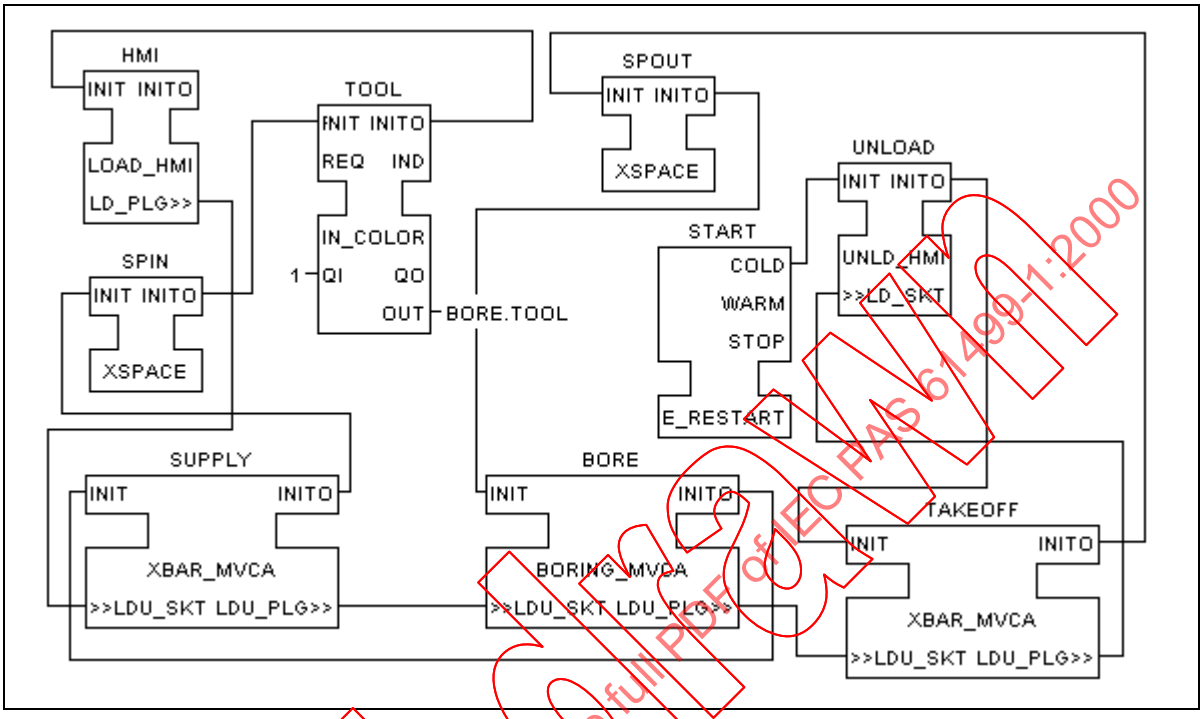


Figure 2.5.2-2 - Illustration of adapter connections

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 H.

2.6. 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 Specification. 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 subclause 3.1.2.

3. SERVICE INTERFACE FUNCTION BLOCKS

This clause defines general principles for the specification of *types* and the behavior of *instances* of *service interface function blocks*, and for two specific types of service interface function blocks, i.e., *communication function blocks* and *management function blocks*.

3.1. General principles

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, the 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 Specification.

3.1.1. 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 3.1.1, as appropriate to the particular service provided. When these are used, their semantics shall be as defined in this clause. The name of the function block *type* shall indicate the provided service.

EXAMPLE - Figures 3.1.1(a) and (b) show examples of service interface function blocks in which the primary interaction is initiated by the *application* and by the *resource*, respectively.

NOTE Some services may provide both resource- and application-initiated interactions in the same service interface function block.

Table 3.1.1 - Standard inputs and outputs for service interface function blocks

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.

Table 3.1.1 - Standard inputs and outputs for service interface function blocks

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). NOTE 1 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 3.1.2. NOTE 2 The function block type specification may define other names for these inputs.
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. NOTE 3 A service specification may indicate that the value of this output is irrelevant for some situations, e.g., for INITO+, IND+ and CNF+ as described in 3.1.2.
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 3.1.2. NOTE 4 The function block type specification may define other names for these outputs.

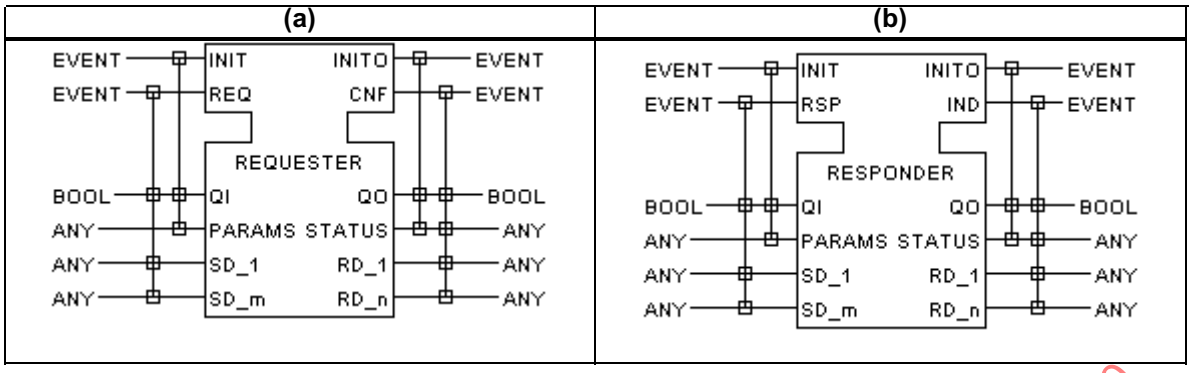


Figure 3.1.1 - Example service interface function blocks:
a) for application-initiated interactions, b) for resource-initiated interactions

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 H for full textual declarations of these function block types.

3.1.2. Behavior of instances

The behavior of *instances* of *service interface function blocks* shall be defined in the corresponding *function block type* specification. This specification can utilize the time-sequence diagrams described in ISO Technical Report 8509. When such diagrams are used, such use shall be subject to the following rules:

1. The normal ISO TR 8509 semantics shall apply, that is:
   a) Time increases in the downward direction.
   b) Events which are sequentially related are linked together across or within resources.
   c) If there is no specific relationship between events, in that it is impossible to foresee which will occur first but both must occur within a finite period of time, a tilde (~) or similar textual notation is used.
2. 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 3.1.2:
   a) 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 3.1.2(a).
   b) 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 3.1.2(b).
3. In the case where the service is represented by two or more service interface function blocks, the notation of figure 3a) of ISO TR 8509 shall be used, as illustrated in Annex F.2.
4. *Service primitives* shall be indicated by horizontal arrows as in ISO TR 8509. 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.
5. 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`.

6. When a Q0 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 Q0 output is TRUE upon the occurrence of the associated event, and the suffix "-" shall be used to indicate that it is FALSE.
7. The standard semantics of asserted (+) and negated (-) events shall be as specified in table 3.1.2.

Figure 3.1.2 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.

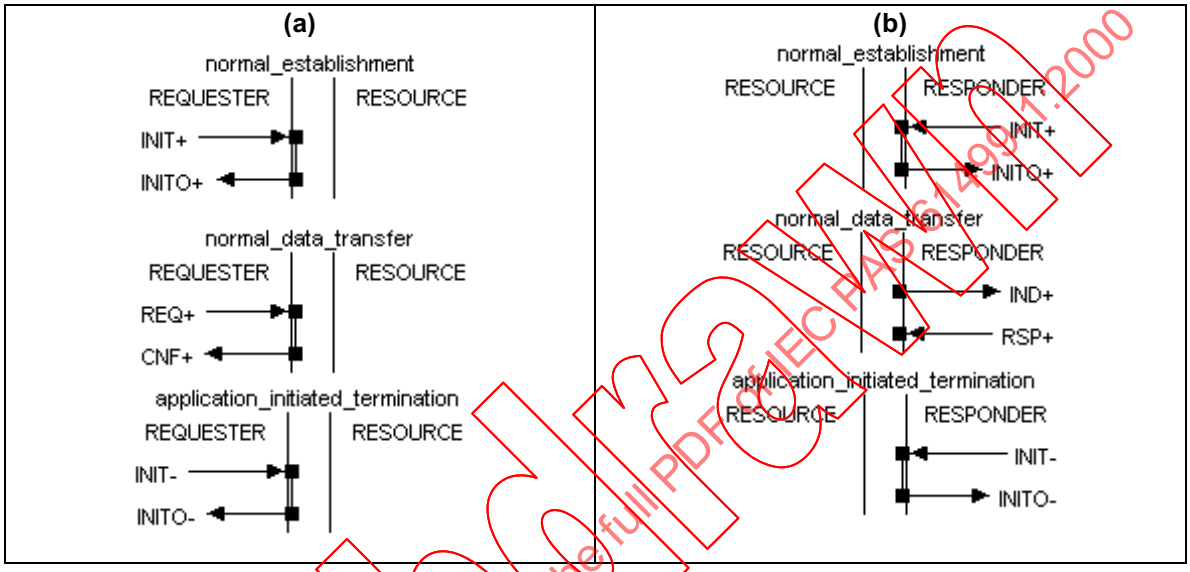


Figure 3.1.2 - Example time-sequence diagrams
a) Diagram for application-initiated (request/confirmation) interactions
b) Diagram for resource-initiated (indication/response) interactions

Table 3.1.2 - 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

3.2. Communication function blocks

Communication function blocks provide *interfaces* between *applications* and the "communication mapping" functions of *resources* as defined in 14.3; hence, they are *service interface function blocks* as described in 3.1.

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* and for the behavior of *instances* of such function block types. Annex F.2 defines generic communication function block types for *unidirectional* and *bidirectional transactions,* and gives rules for the implementation-dependent customization of these types.

3.2.1. Type specification

Declaration of *communication function block types* shall utilize the means defined in subclause 3.1 for the declaration of *service interface function block types*, with the specialized semantics shown in table 3.2.1 for *input* and *output variables*.

Table 3.2.1 - 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 may 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 may be constrained to match the number and types of the corresponding SD inputs.	
[a] *Communication function block type declarations* shall define the number and type of the SD_1,..., SD_m inputs and RD_1,...,RD_n outputs, and may assign them other names.	

3.2.2. Behavior of instances

As illustrated in Annex F.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 subclause 3.1 with the specialized service primitive semantics given in table 3.2.2. Such specification shall include *service primitive* sequences for:

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

Table 3.2.2 - 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

3.3. Management function blocks

This subclause defines requirements and *function block types* for the management of *applications*, and the behaviors of function blocks under the control of *management function blocks*.

3.3.1. Requirements

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

1) In a *resource,* create, initialize, start, stop, delete, query the existence and *attributes* of, and provide notification of changes in availability and status of:
   - *data types*
   - *function block types* and *instances*
   - *connections* among function block instances
2) 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 subclause 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 subclause only deals with item (1) above, i.e., the management of *applications* in *resources*. A framework for device management is described in Annex G.

NOTE 3 The associations among *resources, applications,* and *function block instances* are defined in *system configurations* as described in 4.2.

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

3.3.2. Type specification

Figure 3.3.2-1 illustrates the general form of *management function block types* whose *instances* meet the application management requirements defined above.

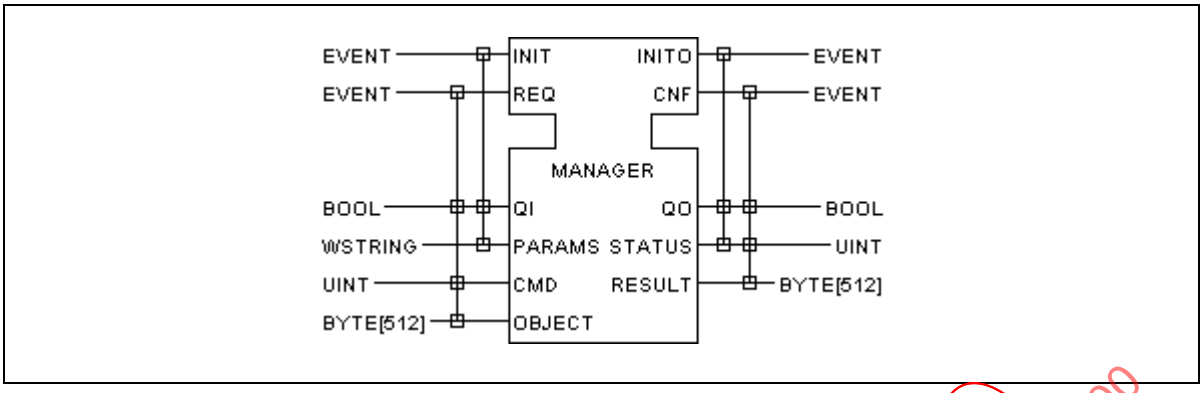


Figure 3.3.2-1 - Generic management function block type

NOTE 1 In particular implementations, the type name (MANAGER in this example) may 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 3.1.

NOTE 3 The INIT and PARAMS inputs and INITO output may or may 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.

The behavior of instances and input/output semantics of management function block types shall follow the rules given in subclause 3.1 for *service interface function block types* with application-initiated interactions, with the additional behaviors shown in figure 3.3.2-2 for unsuccessful service initiation and requests.

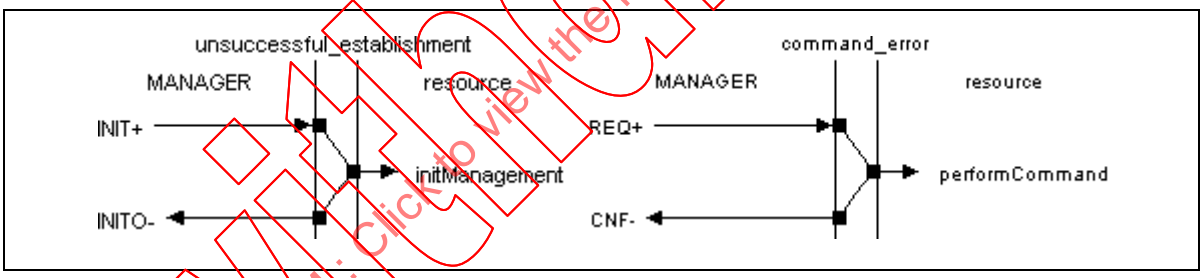


Figure 3.3.2-2 - Service primitive sequences for unsuccessful service

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

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 3.3.2-1.

Table 3.3.2-1 - 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 data from access path
5	WRITE	Write data to access path
6	KILL	Make specified object unrunnable
7	QUERY	Request information on specified object

The values and corresponding semantics of the STATUS output of a management function block shall be as described in Table 3.3.2-2 to express the result of performing the specified command.

Table 3.3.2-2 - 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

Although the The actual lengths of the OBJECT input and RESULT output of management function block instances shall be **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 3.3.2-3. These encodings may use the IEC61499-FBMGT abstract syntax and encoding rules defined in Annex F.3.The actual lengths of these BYTE arrays depend on the particular object and encoding supplied.

NOTE 6 The maximum allowable length of the OBJECT input and RESULT output is an **implementation-dependent parameter**; the value of 512 given in 3.3.2-1 is illustrative.

Table 3.3.2-3 - 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
	access_path_declaration	access_path_name
DELETE	data_type_name	data_type_name
	fb_type_name	fb_type_name
	fb_instance_reference	fb_instance_reference
	connection_definition	connection_definition
	access_path_name	access_path_name
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
	access_path_name	access_path_declaration
READ	access_path_name	accessed_data
WRITE	access_path_data	access_path_name

NOTE 1 See Table 3.3.2-1 for the integer values of the CMD input corresponding to the commands listed above.

NOTE 2 The READ and WRITE commands are limited to a single access path at a time. Other standards may define more complex services, e.g., for multi-variable access, with appropriate service interfaces.

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:

1. START and STOP of a *function block instance* shall be as defined in subclause 3.3.3.
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*.
3. STOP of a *management function block instance* shall be equivalent to STOP of all *function block instances* within the managed *resource.*
4. 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:

1. When the OBJECT input specifies an *event input, event output* or *data output,* the RESULT output shall contain zero or more opposite end points.
2. When the OBJECT input specifies a *data input,* the RESULT output shall list zero or one opposite end point.
3. 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*.

3.3.3. 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 3.3.3-1, subject to the following rules:

1. The capitalized transition conditions in Figure 3.3.3-1 refer to a value of the CMD input, as specified in Table 3.3.2-1, of the management function block upon the occurrence of a REQ+ service primitive.

2. 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 3.3.2-2, under the following conditions:

   - UNSUPPORTED_CMD: No state exists in Figure 3.3.3-1 with a transition condition for the specified CMD value.
   - INVALID_STATE: The currently active state does not have a transition condition for the specified CMD value.
   - 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.
   - 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.

3. 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 3.3.2-2, and a corresponding value for the `RESULT` output as defined in Table 3.3.2-3.

4. The semantics of the actions shown in Figure 3.3.3-1 shall be as follows:

   - **`runECinputs`:** While a state containing this action is active, operation of the state machine shown in Figure 2.2.2.2-1 is enabled for all event inputs. If the managed function block is a *service interface*, invocation of service primitives by events at the event inputs is enabled. If the managed function block is a *service interface*, *invocation* of *service primitives* by events at the event inputs is enabled.
   - **`runECC`:** While a state containing this action is active, operation of the ECC state machine shown in Figure 2.2.2.2-2 is enabled. If the managed function block is a *service interface*, generation of events corresponding to service primitives at the event inputs is enabled.
   - **`stopAlgorithm`:** *Execution* of the currently active *algorithm* (if any) is terminated immediately. If the managed function block is a *service interface*, service *operations* are terminated immediately.
   - **`completeAlgorithm`:** *Execution* of the currently active *algorithm* (if any) is allowed to complete. If the managed function block is a *service interface*, the currently active service primitive is allowed to complete.
   - **`initialize`:** Forces all event inputs and the execution control chart (ECC) state machine to their respective initial (s0) states. All variables are initialized to their initial values as defined in 2.2.2.1. If the managed function block is a *service interface*, the service is placed in a state in which it can perform a normal initialization sequence in response to an `INIT+` primitive as described in 3.1.2.

5. The actions described in the previous rule apply recursively to all *component function blocks* of a managed *composite function block*.

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 may be specified by the manufacturer of a compliant device, for example by reference to an appropriate Standard.

NOTE 3 *Applications* may 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.

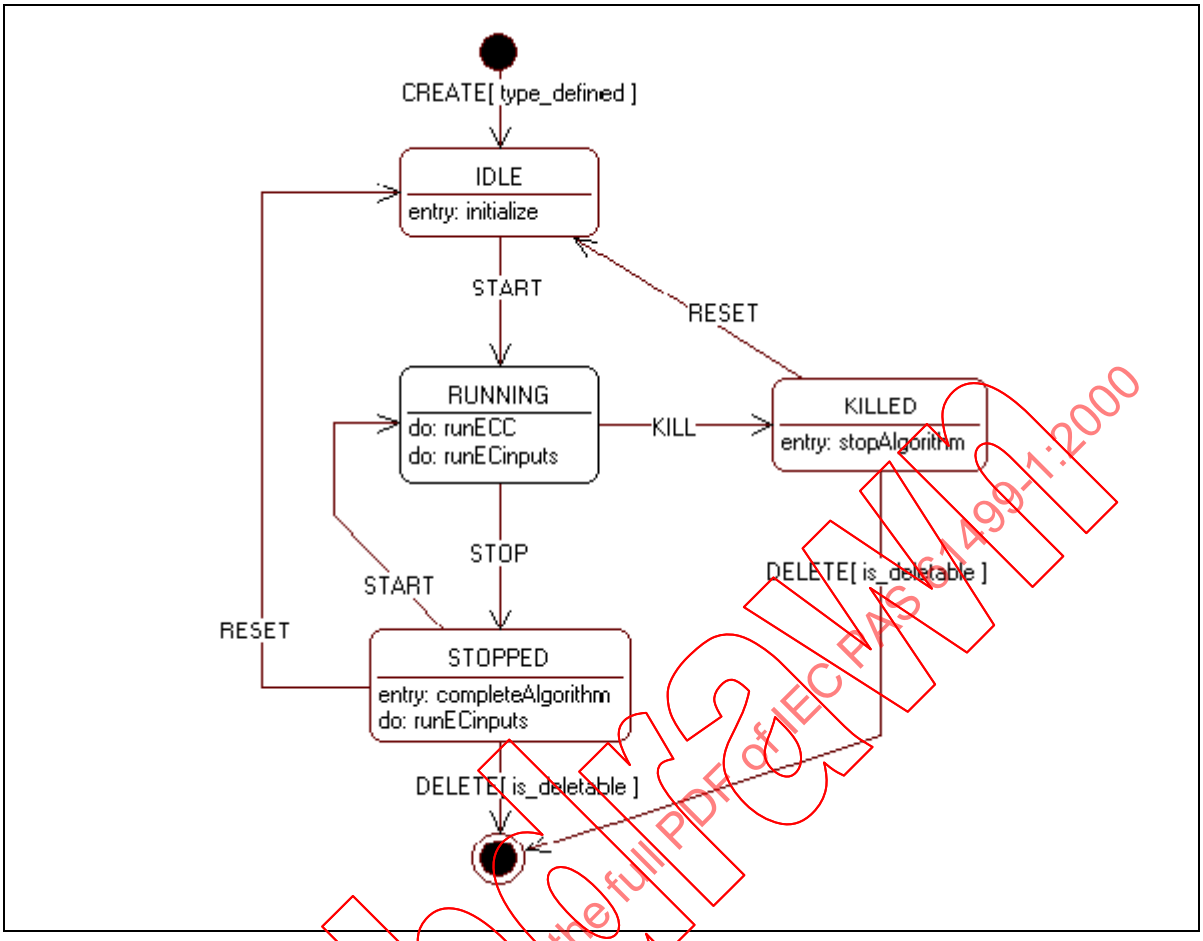


Figure 3.3.3-1 - Operational state machine of a managed function block

As described in 2.4.2, execution control in *subapplications* is entirely deferred to the execution control mechanisms of their *component function blocks* and *component subapplications*. However, the rules given in 2.3.2 for the behavior of instances of *composite function block types* do not explicitly specify the relationships between the time of *sampling* of the *input variables* and the execution control of the component function blocks. This could lead to indeterminacy in the operation of such blocks, particularly when they contain *service interface function blocks* as *component function blocks*.

Such indeterminacy is avoided by substituting the RUNNING state of Figure 3.3.3-2 for the RUNNING state of Figure 3.3.3-1. The explanation of the states and transitions in this diagram is given in Table 3.3.3.

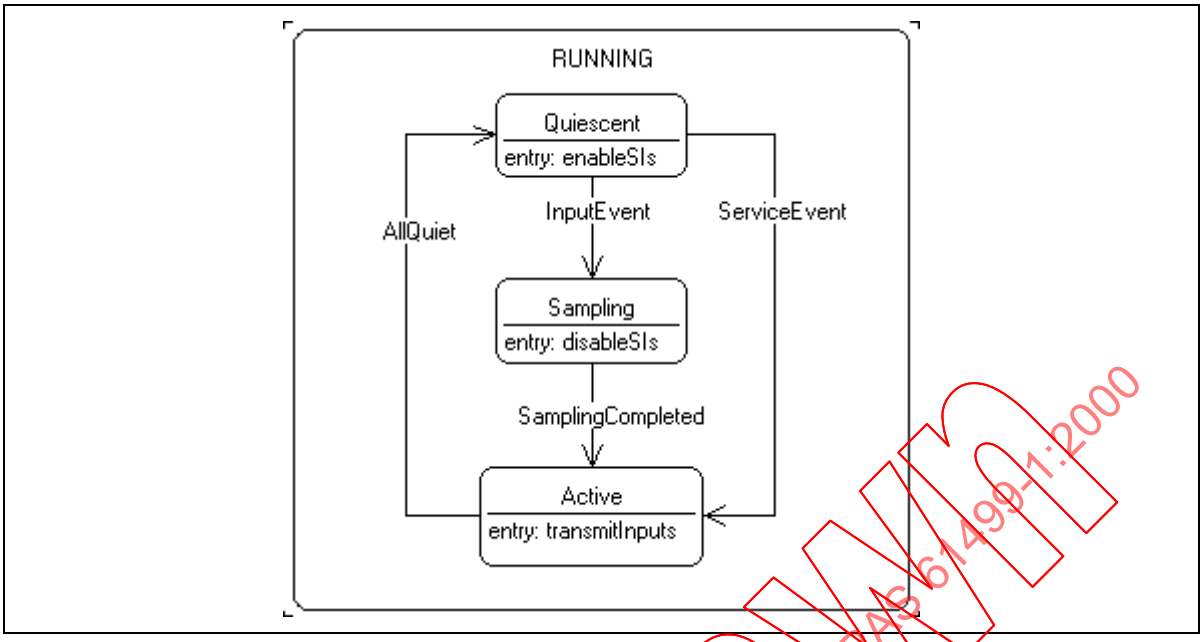


Figure 3.3.3-2 - RUNNING state for composite function blocks

Table 3.3.3 - Substates, transitions and actions of Figure 3.3.3-2

State	Description
Quiescent	All component function blocks are in the **Quiescent** state and no events are pending at event inputs.
Sampling	Input variables are being sampled corresponding to all active event inputs of the composite function block.
Active	Propagation of events and data within the function block is enabled.
Transition	**Condition**
InputEvent	An event has arrived on at least one event input.
ServiceEvent	At least one event is pending at an output of a component service interface function block.
SamplingCompleted	Sampling of all input variables corresponding to active event inputs has been completed.
AllQuiet	All component function blocks have returned to their **Quiescent** states.
Action	**Description**
enableSIs	Enable component service interface function blocks to request generation of events at their event outputs.
disableSIs	Disable component service interface function blocks from requesting generation of events at their event outputs.
transmitInputs	Transmit events from all active event inputs of the composite function block to their corresponding destinations.

4. CONFIGURATION OF FUNCTIONAL UNITS AND SYSTEMS

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

1. An IPMCS consists of interconnected *devices.*
2. A *device* is an *instance* of a corresponding device *type*.
3. The functional capabilities of a *device type* are described in terms of its associated *resources.*
4. A *resource* is an *instance* of a corresponding resource *type*.
5. 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*. This clause defines the following sets of rules to support this process:

- Subclause 4.1 defines rules for the functional specification of *types* of *resources* and *devices.*
- Subclause 4.2 defines rules for the *configuration* of an IPMCS in terms of its associated *devices* and *applications.*

4.1. Functional specification of types

This subclause defines rules for the functional specifications of *types* of *devices* and *resources*.

4.1.1. Functional specification of resource types

The functional specification of a *resource type* shall include:

- 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 maximum numbers of *data connections* and *event connections* that can exist in an instance of the resource type;
- the instance names, types, and initial values of any function block instances that are automatically present in each instance of the resource type;
- any *data connections* and *event connections* that are automatically present in each instance of the resource type;
- any *access paths* that are automatically present in each instance of the resource type.

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

- the time (identified as "T_{alg}" in figure 1.4.5.3-2) 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" may 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 Specification except as such interfaces are represented by *service interface function blocks*.

4.1.2. Functional specification of device types

The functional specification of a *device type* shall include:

- the device *type name*;
- the *instance name, data type*, and initialization of each of the device *parameters*;
- the instance name, type name, and initialization of each *function block instance* that is automatically present in each *instance* of the device type;
- any *data connections* and *event connections* that are automatically present in each instance of the device type;
- 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 automatically present in the resource instance in each instance of the device type;
  3. any *data connections* and *event connections* that are automatically present in the resource instance in each instance of the device type;
  4. any *access paths* that are automatically present in the resource instance in each instance of the device type.

NOTE 1 - Items (2), (3) and (4) above are considered to be in addition to the corresponding elements declared in the resource type specification as defined in subclause 4.1.1.

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 document except as such interfaces are represented by *service interface function blocks*.

4.2. Configuration requirements

This subclause defines rules for the *configuration* of industrial process measurement and control *systems, devices, resources*, and *applications*.

4.2.1. Configuration of systems

The configuration of a *system* shall include:

- the *name* of the system;
- the specification of each *application* in the system, as specified in subclause 4.2.2;
- the configuration of each *device* and its associated *resources*, as specified in subclause 4.2.3.

4.2.2. Specification of applications

The specification of an *application* shall consist of:

- its name in the form of an *identifier*;
- the *instance name, type name, data connections* and *event 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*.

4.2.3. Configuration of devices and resources

The configuration of a *device* shall consist of:

1) the *instance name* and *type name* of the device;

2) configuration-specific values for the device *parameters*;
3) the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
4) 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;
5) any *data connections* and *event connections* that are present in the device instance in addition to those defined for the device type;
6) the *resource types* supported by the device *instance* in addition to those specified for the device *type*;
7) 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*.

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* shall consist of:

1) its *instance name* and *type name*;
2) the *data types* and *function block types* supported by the resource *instance;*
3) the *instance name, type name*, and initialization of each function block instance that is present in the resource instance;
4) any *data connections, event connections* and *adapter connections* that are present in the resource instance;
5) any *access paths* that are present in the resource instance.

Resource configuration shall be subject to the following rules:

a. The name of a function block *instance* allocated to a *resource* from an application shall consist of the application name concatenated to the function block instance name within the application with a period (" . "), for example `APP1.FB2`. If *subapplications* are instantiated in the application, the subapplication instance name (or names, if more than one subapplication is nested) shall be inserted in the concatenation before the respective function block instance name.

b. Items (2), (3), (4) and (5) above are considered to be in addition to the corresponding elements declared in the device and resource type specifications as defined in subclauses 4.1.2 and 4.1.1, respectively.

c. Items (3) and (4) include *function block instances, data connections* and *event connections* from those portions of *applications* allocated to the resource.

d. Items (3) and (4) 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*.

e. The items in Item (3) 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 4.1.1.

f. It shall be an **error** if:

- the instance name of a resource is not unique within the scope of the device containing it;
- 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.

5. COMPLIANCE

This clause contains rules for compliance with the provisions of this Part of this Specification. Compliance requirements are defined for industrial process measurement and control *systems, devices*, and standards.

5.1. Compliant systems and subsystems

A *system* which is compliant with this Part may provide features not specified in this Part. Such features shall be described in a document which identifies them as "extensions to IEC 61499-1" and specifies their *mapping* to the elements described in this Part.

5.2. Compliant devices

Types and *instances* of *devices* complying with this Part of this specification shall be specified according to the rules given in clause 3.

Devices complying with this Part of this specification shall be characterized as belonging to one of three compliance classes, namely:

- Class 0: Simple devices
- Class 1: Simple programmable devices
- Class 2: User-reprogrammable devices

This compliance is specified in terms of the management commands to which each type of device can respond, as given in table 5.2.

Table 5.2 - Device compliance classes

CMD [a]	OBJECT	CLASS 0	CLASS 1	CLASS 2
CREATE	type_declaration			Required
	fb_type_declaration			Required
	fb_instance_definition		Required	Required
	connection_definition	Required [b]	Required	Required
	access_path_declaration		Required [c]	Required [c]
DELETE	data_type_name			Required
	fb_type_name			Required
	fb_instance_reference		Required	Required
	connection_definition		Required	Required
	access_path_name		Required [c]	Required [c]
START	fb_instance_reference	Required	Required	Required
	application_name	Required	Required	Required
STOP	fb_instance_reference	Required	Required	Required
	application_name	Required	Required	Required
KILL	fb_instance_reference		Required	Required
QUERY	all_data_types	Required	Required	Required
	all_fb_types	Required	Required	Required
	data_type_name			Required
	fb_type_name			Required
	fb_instance_reference		Required	Required
	connection_start_point		Required	Required
	application_name		Required	Required
	access_path_name		Required [c]	Required [c]
READ	access_path_name	Required [c]	Required [c]	Required [c]
WRITE	access_path_data	Required [c]	Required [c]	Required [c]

[a] See 3.3.2 for definition of the semantics of these commands

[b] Only connection of a new *parameter* value to a *data input* of a function block is required of Class 0 devices.

[c] This capability is required only in devices that support *access paths*.

5.3. Compliant standards

A standard which is compliant with this Part shall include specifications of:

- the kinds of *applications* addressed by the compliant standard;
- the functional scope addressed by the compliant standard, i.e., *system, device* or *resource*;
- the *function block types* to be used to implement the applications within the functional scope of the compliant standard, in the format defined in clause 2 or clause 3;
- the means to be used to meet the requirements in clause 4 for *configuration* of *entities* within the functional scope of the compliant standard;
- the means to be used to meet the requirements of subclause 3.2 for communication among entities within the functional scope of the compliant standard;
- the means to be used to meet the requirements of subclause 3.3 for management of entities within the functional scope of the compliant standard.

A compliant standard shall also specify:

- any additional requirements beyond those of this Part which must be met within the scope of the compliant standard;
- any requirements of this Part which need not be met within the scope of the compliant standard.

Furthermore, if the means used to specify the provisions of a compliant standard differ from those defined in this Part, the compliant standard shall define:

- the *mapping* between such means in the compliant standard and those defined in this Part;
- any other such means employed in the compliant standard beyond those defined in this Part.

Furthermore, if any such exceptions or extensions are identified, the appropriate IEC Committee or Subcommittee shall be notified of the possible need for modifications of this Part.

A compliant standard may require that the graphic representations defined in this Part be utilized, or may specify alternative graphical, textual or tabular representations provided that it specifies an unambiguous *mapping* to the non-terminal symbols defined in Annex B of this Part and their associated textual or graphical semantics.

ANNEX A - EVENT FUNCTION BLOCKS (normative)

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 this Annex 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 this Annex 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 1.4.5.3:

- The parameters **Tsetup**, **Tstart** and **Tfinish** are considered to be zero (0) for all *component function blocks* in the reference implementation.
- The parameter **Talg** 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 this Annex are *service interface function block types*.

Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

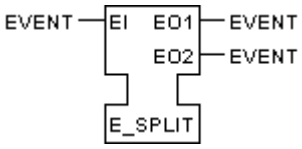
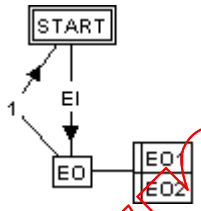
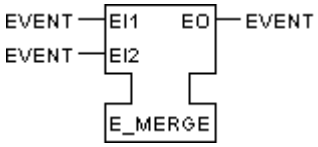
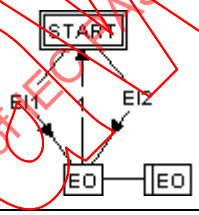
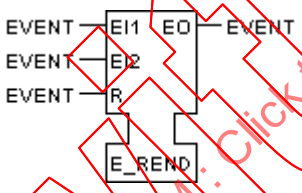
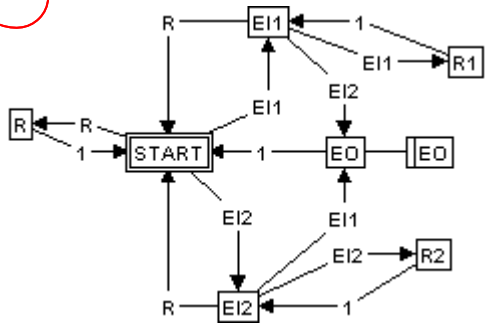
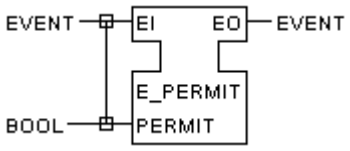
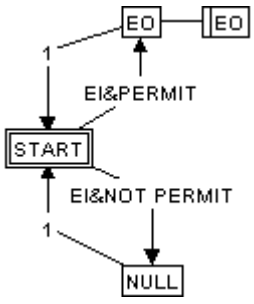
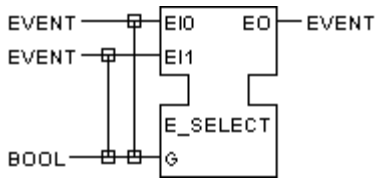
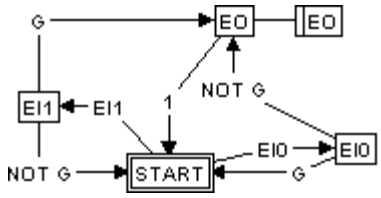
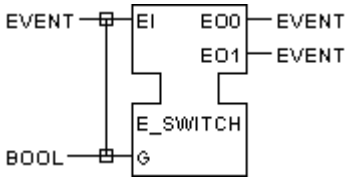
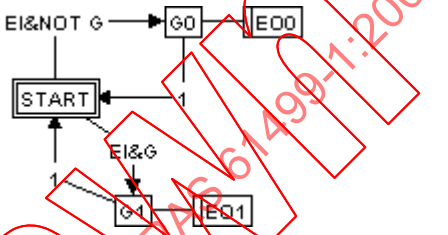
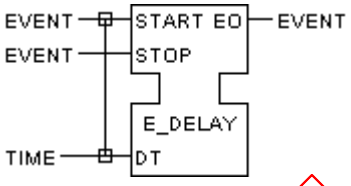
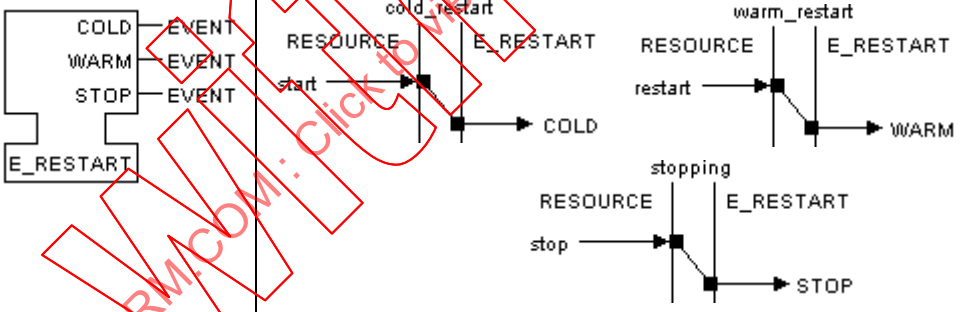
No.	Description	
	Interface	**ECC/Algorithms/Service sequences**
1	**Split an event**	
		
The occurrence of an event at EI causes the occurrence of events at EO1, EO2, . . . , EOn (n=2 in the above example).		
2	**Merge (OR) of multiple events**	
		
The occurrence of an event at any of the inputs EI1, EI2, . . . , EIn causes the occurrence of an event at EO (n=2 in the above example).		
3	**Rendezvous of two events**	
		
4	**Permissive propagation of an event**	
		

Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

5	**Selection between two events**
	
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.
8	**Generation of restart events**
	

1. An event is issued at the COLD output upon "cold restart" of the associated resource.
2. An event is issued at the WARM output upon "warm restart" of the associated resource.
3. An event is issued at the STOP output (if possible) prior to "stopping" of the associated resource.

NOTE - See IEC 61131-3 for a discussion of "cold restart" and "warm restart".

Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

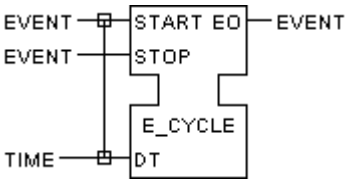
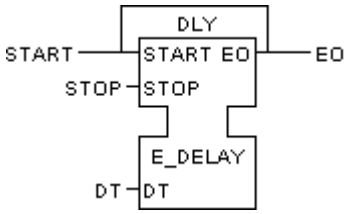
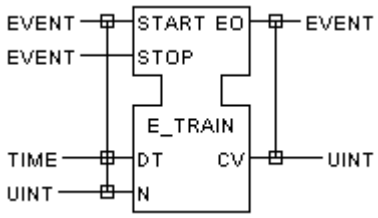
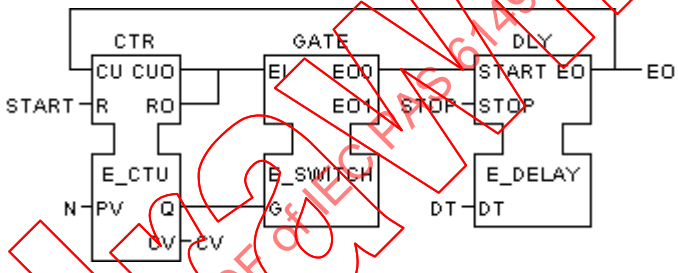
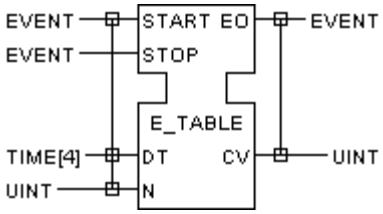
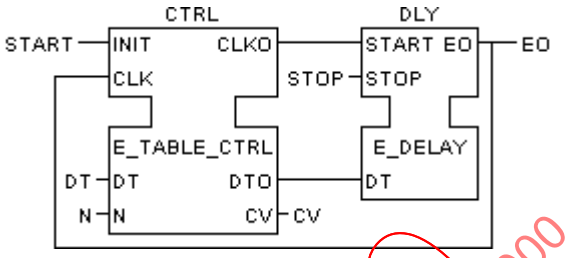
9	**Periodic (cyclic) generation of an event**
	
An event occurs at EO at an interval DT after the occurrence of an event at START, and at intervals of DT thereafter until the occurrence of an event at STOP.	
10	**Generation of a finite train of events**
	
	NOTE - See table entry #18 for a definition of the E_CTU type.
An event occurs at EO at an interval DT after the occurrence of an event at EI, and at intervals of DT thereafter, until N occurrences have been generated or an event occurs at the STOP input.	

Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

11	Generation of a finite train of events (table driven)

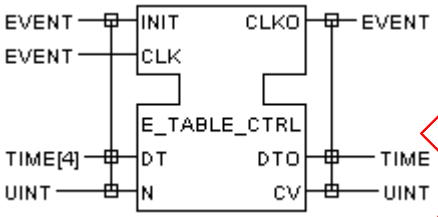


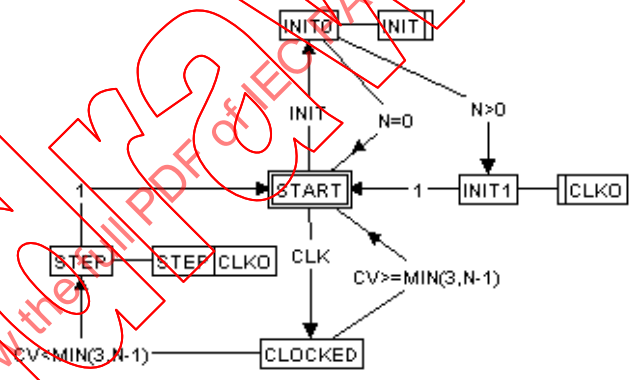


An event occurs at `EO` at an interval `DT[0]` after the occurrence of an event at `EI`. A second event occurs at an interval `DT[1]` after the first, etc., until `N` occurrences have been generated or an event occurs at the `STOP` input. The current event count is maintained at the `CV` output.

NOTE 1 - In this example implementation, `N <= 4`.

NOTE 2 - Implementation using the `E_TABLE_CTRL` function block type illustrated below is not a normative requirement. Equivalent functionality may be implemented by various means.





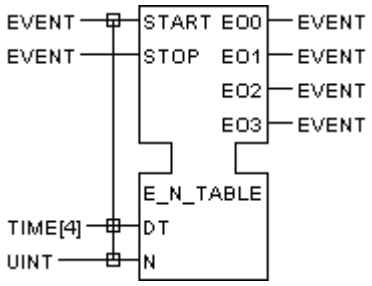

```
ALGORITHM INIT IN ST:
  CV := 0 ;
  DTO := DT[0];
END_ALGORITHM
```

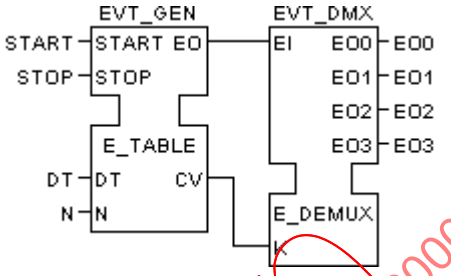
```
ALGORITHM STEP IN ST:
  CV := CV+1;
  DTO := DT[CV];
END_ALGORITHM
```

Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

12	**Generation of a finite train of separate events (table driven)**

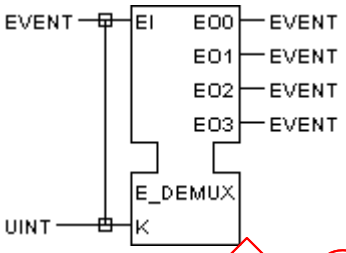


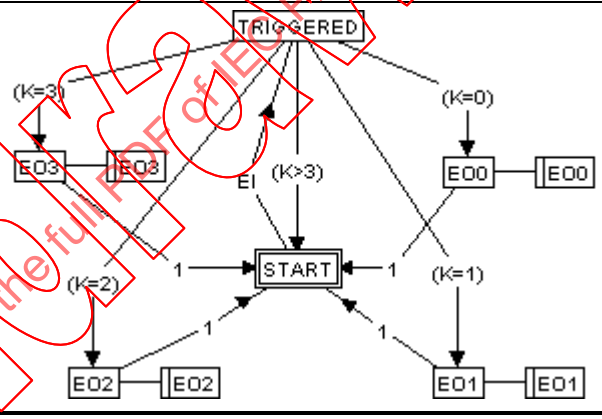


An event occurs at `EO0` at an interval `DT[0]` after the occurrence of an event at `EI`. An event occurs at `EO2` an interval `DT[1]` after the occurrence of the event at `EO1`, etc., until `N` occurrences have been generated or an event occurs at the `STOP` input.

NOTE 1 - In this example implementation, `N <= 4`.

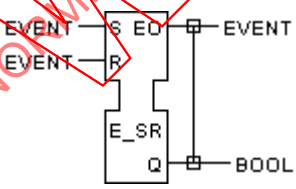
NOTE 2 - Implementation using the `E_DEMUX` function block type illustrated below is not a normative requirement. Equivalent functionality may be implemented by various means.

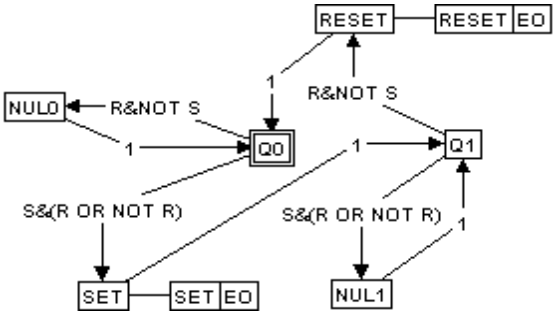




13	**Event-driven bistable (Set dominant)**

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. If simultaneous `S` and `R` events occur, the `S` input is dominant. An event is issued at the `EO` output when the value of `Q` changes.





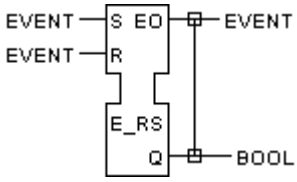

```
ALGORITHM SET IN ST :  (* Set Q *)
  Q := TRUE ;
END_ALGORITHM
```

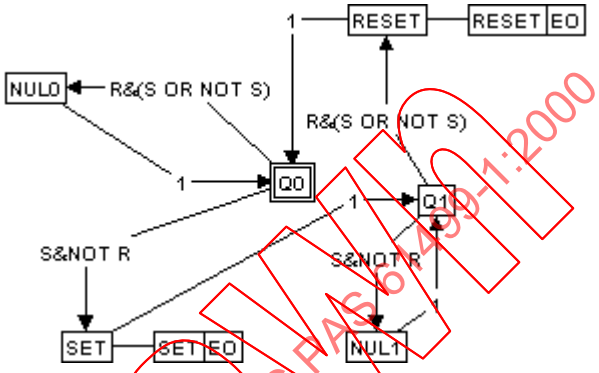
```
ALGORITHM RESET IN ST : (* Reset Q *)
  Q := FALSE ;
END_ALGORITHM
```


Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

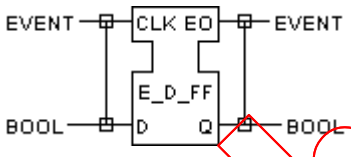
14	**Event-driven bistable (Reset dominant)** The output `Q` is set to `1 (TRUE)` upon the occurrence of an event at the `S` input, and is reset to `0 (FALSE)` upon the occurrence of an event at the `R` input. If simultaneous `S` and `R` events occur, the `R` input is dominant. An event is issued at the `EO` output when the value of `Q` changes.



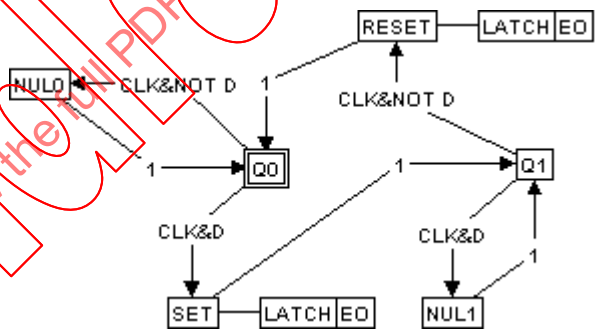


NOTE - Algorithms SET and RESET are the same as for E_SR.

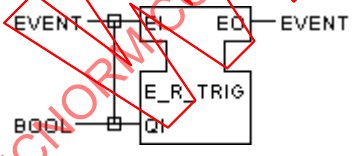
15	**D (Data latch) bistable**

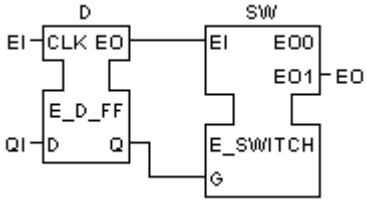



```
ALGORITHM LATCH IN ST :
 Q := D ;
END_ALGORITHM
```

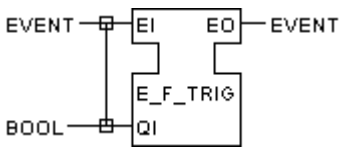


16	**Boolean rising edge detection**





17	**Boolean falling edge detection**



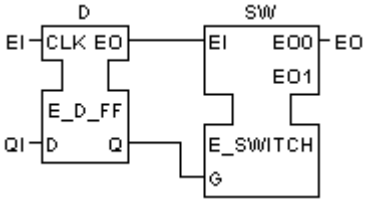
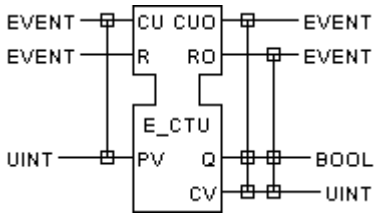
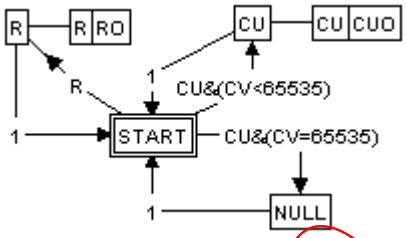


Table A.1 - Event function blocks

NOTE - Full textual specifications of all function block types shown in this table are given in Annex H.

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.1(b) is equivalent to the explicit representation in figure A.1(a).

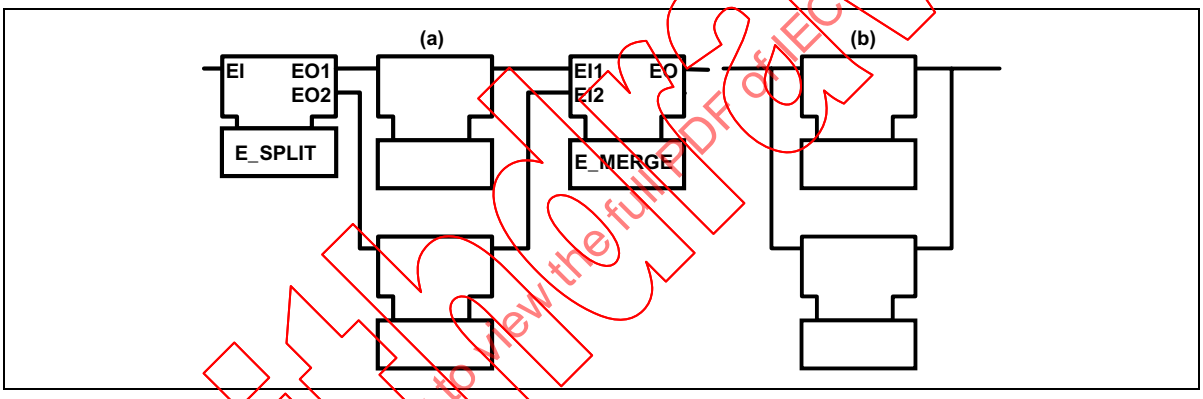


Figure A.1 - Event split and merge
a) Explicit representation, b) Implicit representation

NOTE - Irrelevant detail is suppressed in the above figure.

ANNEX B - TEXTUAL SYNTAX (normative)

B.1. Syntax specification technique

The textual constructs in this Annex 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.

B.1.1. Syntax

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

B.1.1.1. Terminal symbols

The terminal symbols for textual specifications of entities defined in this Part consist of combinations of the characters in the "ISO-646 IRV" given as Table 1 - Row 00 of ISO/IEC 10646.

For the purposes of this Specification, 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
```

B.1.1.2. Non-terminal symbols

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.

B.1.1.3. Production rules

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

```
non_terminal_symbol ::= extended_structure
```

This rule can be read as:

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

Extended structures can be constructed according to the following rules:

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

2) A terminal symbol is an extended structure.

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

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

`(S)`, meaning S itself.

`{S}`, *closure*, meaning zero or more concatenations of S.

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

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

`S1 | S2`, *alternation*, meaning a choice of `S1` or `S2`.

`S1 S2`, *concatenation*, meaning `S1` followed by `S2`.

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

B.1.2. Semantics

Semantics are defined in this Specification 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 this subclause can be used for the textual specification of *function block types* according to the rules given in clauses 2 and 3 of this Part.

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]   <only for service interface FB>
     '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 Annex E.1>
```

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

fb_instance_definition ::= fb_instance_name ':' fb_type_name

plug_list ::= 'PLUGS'
     {plug_name {',' plug_name} ':' adapter_type_name ';'}
     'END_PLUGS'

socket_list ::= 'SOCKETS'
     {socket_name {',' socket_name} ':' adapter_type_name ';'}
     '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 ::=
     ((fb_instance_name '.' event_output_name)
              | (plug_name '.' event_input_name))
       'TO' ((fb_instance_name '.' event_input_name)
             | (plug_name '.' event_output_name)) ';'
     | event_input_name 'TO'
             ((fb_instance_name '.' event_input_name)
            | (plug_name '.' event_output_name)) ';'
     | ((fb_instance_name '.' event_output_name)
             | (plug_name '.' event_input_name))
       'TO' event_output_name ';'

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

data_conn ::=
     ( fb_instance_name '.' output_variable_name
             | plug_name '.' input_variable_name
             | input_variable_name | constant)
       'TO' ((fb_instance_name '.' input_variable_name)
             | (plug_name '.' output_variable_name)) ';'
     | ((fb_instance_name '.' output_variable_name)
             | (plug_name '.' input_variable_name))
       'TO' output_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 | '->' event_output_name
     | algorithm_name '->' event_output_name

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

ec_transition_condition ::=
     <Boolean expression as defined in compliant standards>
     <May utilize input, output, internal and event variables>

fb_algorithm_declaration ::=
     'ALGORITHM' algorithm_name 'IN' language_type ':'
      algorithm_body
     'END_ALGORITHM'

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 clause 2 of this Part.

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]
          [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]

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

subapp_event_conn ::=
     (fb_subapp_name '.' event_output_name
       'TO' fb_subapp_name '.' event_input_name ';')
     | (event_input_name 'TO' fb_subapp_name '.' event_input_name ';')
     | (fb_subapp_name '.' event_output_name 'TO' event_output_name ';')

fb_subapp_name ::= fb_instance_name | subapp_instance_name
```

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

subapp_data_conn ::=
     ((( fb_subapp_name '.' output_variable_name )
       | input_variable_name | constant)
       'TO' fb_subapp_name '.' input_variable_name ';')
     | ((fb_subapp_name '.' output_variable_name)
       'TO' output_variable_name ';')

subapp_type_name ::= identifier

subapp_instance_name ::= identifier
```

B.3. Configuration elements

The syntax defined in this subclause can be used for the textual specification of *resource types, device types, applications,* and *system configurations* according to the rules given in clause 4 of this Part.

The productions given in B.2 also apply to this subclause.

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}
      [device_param_list]
      [mappings]
     'END_SYSTEM'

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

device_param_list ::= 'DATA_CONNECTIONS'
      {device_param_conn}
     'END_CONNECTIONS'

device_param_conn ::=
    constant 'TO' device_name '.' input_variable_name ';'

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

resource_configuration ::=
     'RESOURCE' resource_instance_name ':' resource_type_name
      [fb_type_list]
      [config_fb_instance_list]
      [config_connection_list]
      [access_paths]
     'END_RESOURCE'

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

```
config_fb_instance_list ::= 'FBS' {config_fb_instance} 'END_FBS'

config_fb_instance ::= fb_instance_ref_definition ';'

fb_instance_ref_definition ::= fb_instance_reference ':' fb_type_name

fb_instance_reference ::= [app_hierarchy_name] fb_instance_name

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

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_reference '.' event_output_name
      'TO' fb_instance_reference '.' event_input_name ';'

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

config_data_conn ::=
     ((fb_instance_reference '.' output_variable_name) | constant)
      'TO' fb_instance_reference '.' input_variable_name ';'

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

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

access_paths ::= 'VAR_ACCESS'
     access_path_declaration {access_path_declaration}
     'END_VAR'

access_path_declaration ::=
     access_path_name ':' access_path [access_direction] ';'

access_path ::=
     fb_instance_reference {'.' fb_instance_name} '.' symbolic_variable
      <symbolic_variable is defined in IEC 61131-3, Annex B.1.4>

access_direction ::= 'READ_ONLY' | 'READ_WRITE'
     <default is READ_ONLY>
     <READ_WRITE only applies to unconnected input variables or internal
     variables>

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]
      [devtype_connection_list]
      [devtype_access_paths]
     'END_DEVICE_TYPE'

devtype_connection_list ::=
     [devtype_event_conn_list]
     [devtype_data_conn_list]
     [devtype_adapter_conn_list]
```

```
devtype_event_conn_list ::=
     'EVENT_CONNECTIONS'
      {devtype_event_conn}
     'END_CONNECTIONS'

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

devtype_data_conn_list ::=
     'DATA_CONNECTIONS'
      {devtype_data_conn}
     'END_CONNECTIONS'

devtype_data_conn ::=
     (fb_instance_name '.' output_variable_name
            | input_variable_name | constant)
      'TO' fb_instance_name '.' input_variable_name ';'

devtype_adapter_conn_list ::= 'ADAPTER_CONNECTIONS'
      {devtype_adapter_conn}
     'END_CONNECTIONS'

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

devtype_access_paths ::= 'VAR_ACCESS'
     {devtype_access_path_declaration}
     'END_VAR'

devtype_access_path_declaration ::=
     access_path_name ':' devtype_access_path [access_direction] ';'

devtype_access_path ::=
     fb_instance_name ['.' fb_instance_name] '.' symbolic_variable
     <symbolic_variable is defined in IEC 61131-3, Annex B.1.4>

resource_instance ::=
     'RESOURCE' resource_instance_name ':' resource_type_name
      [fb_instance_list]
      [devtype_connection_list]
      [devtype_access_paths]
     '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]
      devtype_connection_list
      [devtype_access_paths]
     'END_RESOURCE_TYPE'

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

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

fb_resource_reference = resource_hierarchy '.' fb_instance_name

resource_hierarchy ::= device_name ['.' resource_instance_name]

system_name ::= identifier

device_name::= identifier

device_type_name::= identifier

application_name ::= identifier

resource_instance_name::= identifier

resource_type_name::= identifier

access_path_name ::= identifier
```

B.4. Common elements

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

B.5. Supporting productions for management commands

The syntax defined in this subclause is referenced in tables 3.3.2-3 and 5.2.

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

access_path_data ::= access_path_name ':' accessed_data

accessed_data ::= data_element {',' data_element}

data_element ::= constant | enumerated_value | structure_initialization
    | array_initialization

all_data_types ::= 'ALL_DATA_TYPES'

all_fb_types ::= 'ALL_FB_TYPES'

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

B.6. Tagged data types

SYNTAX:

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

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

SEMANTICS

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

B.7. Adapter interface types

SYNTAX:

```
adapter_type_declaration ::=
    'ADAPTER' adapter_type_name
        fb_interface_list
    'END_ADAPTER'

adapter_type_name ::= identifier
```

SEMANTICS: See 2.5.

ANNEX C - OBJECT MODELS (informative)

This Annex 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 Part.

The notation used in this Annex 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.rational.com/uml/** .

NOTE: In this Annex, object class String is used in Table C.2.1 to model implementation-dependent objects, such as the source and destination of data and event connections, which may or may not be implemented as strings.

C.1. ESS Models

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.

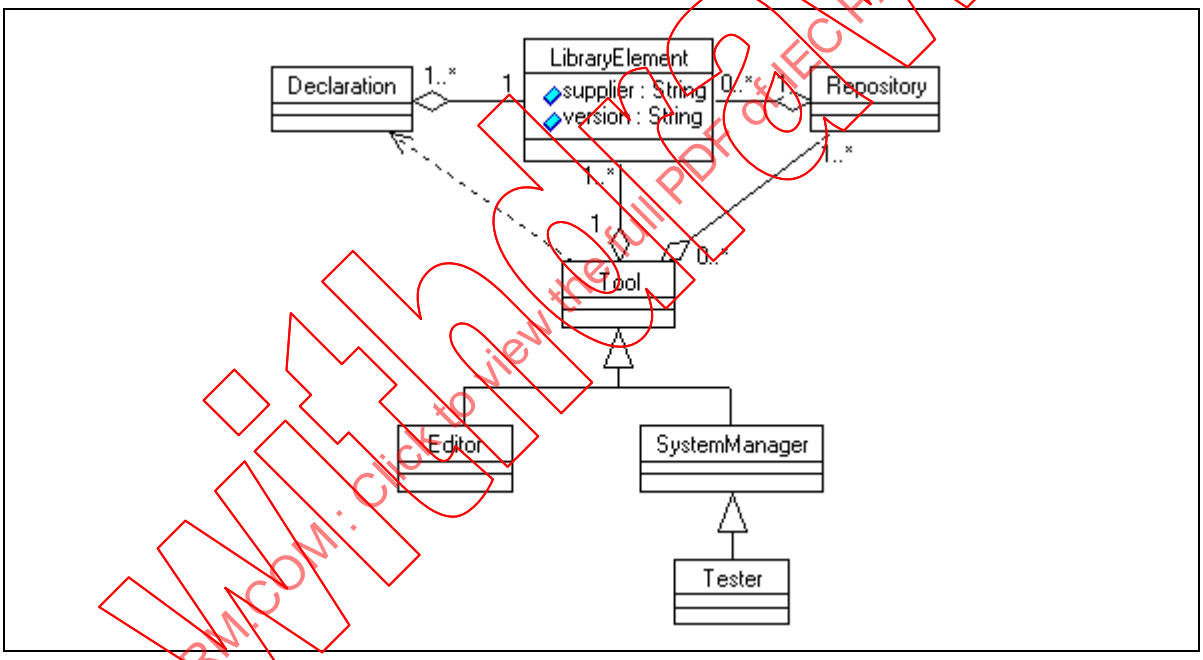


Figure C.1 - ESS Overview

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 IEC 61499-1. 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.1.2) 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.1.1 Library elements

The subclasses of **LibraryElement** are shown in Figure C.1.1. The syntactic production in Annex B of IEC 61499-1 corresponding to each subclass is listed in Table C.1.1.

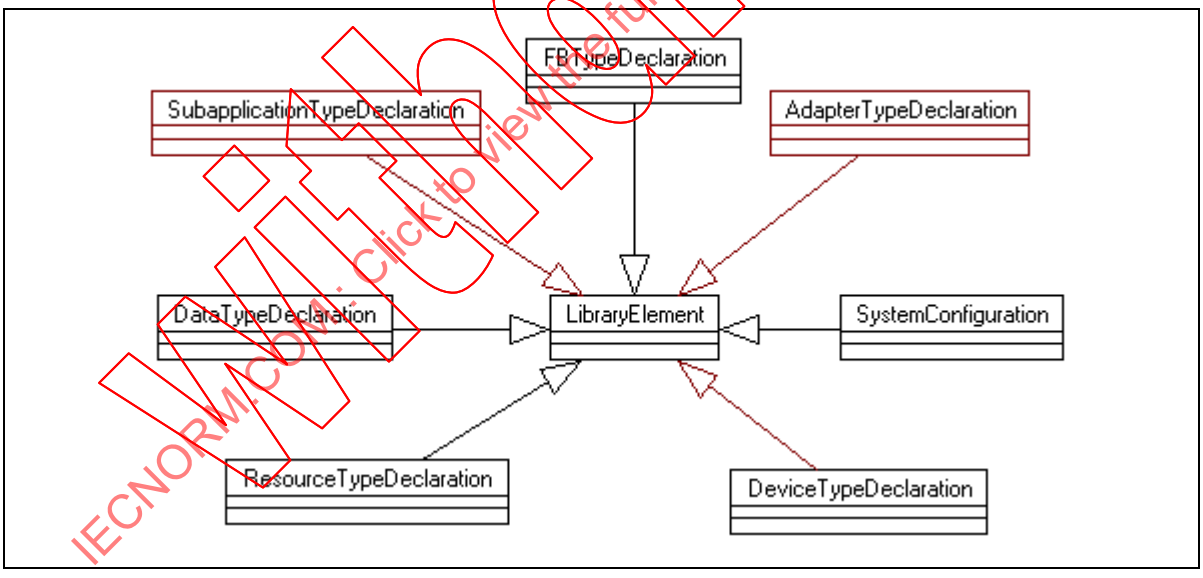


Figure C.1.1 - Library elements

Table C.1.1 - Syntactic productions for library elements

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

C.1.2 Declarations

Figure C.1.2 shows the class hierarchy of *declarations* which may be manipulated by *software tools*. The syntactic productions in Annex B of IEC 61499-1 corresponding to each of these subclasses are listed in Table C.1.2.

NOTE: The operations **fromString()** and **toString()** for class Declaration and its subclasses represent the parsing of a new instance of the corresponding class from a string in the syntax specified in Annex B of IEC 61499-1 and the generation of a string representing the instance in the corresponding syntax, respectively.

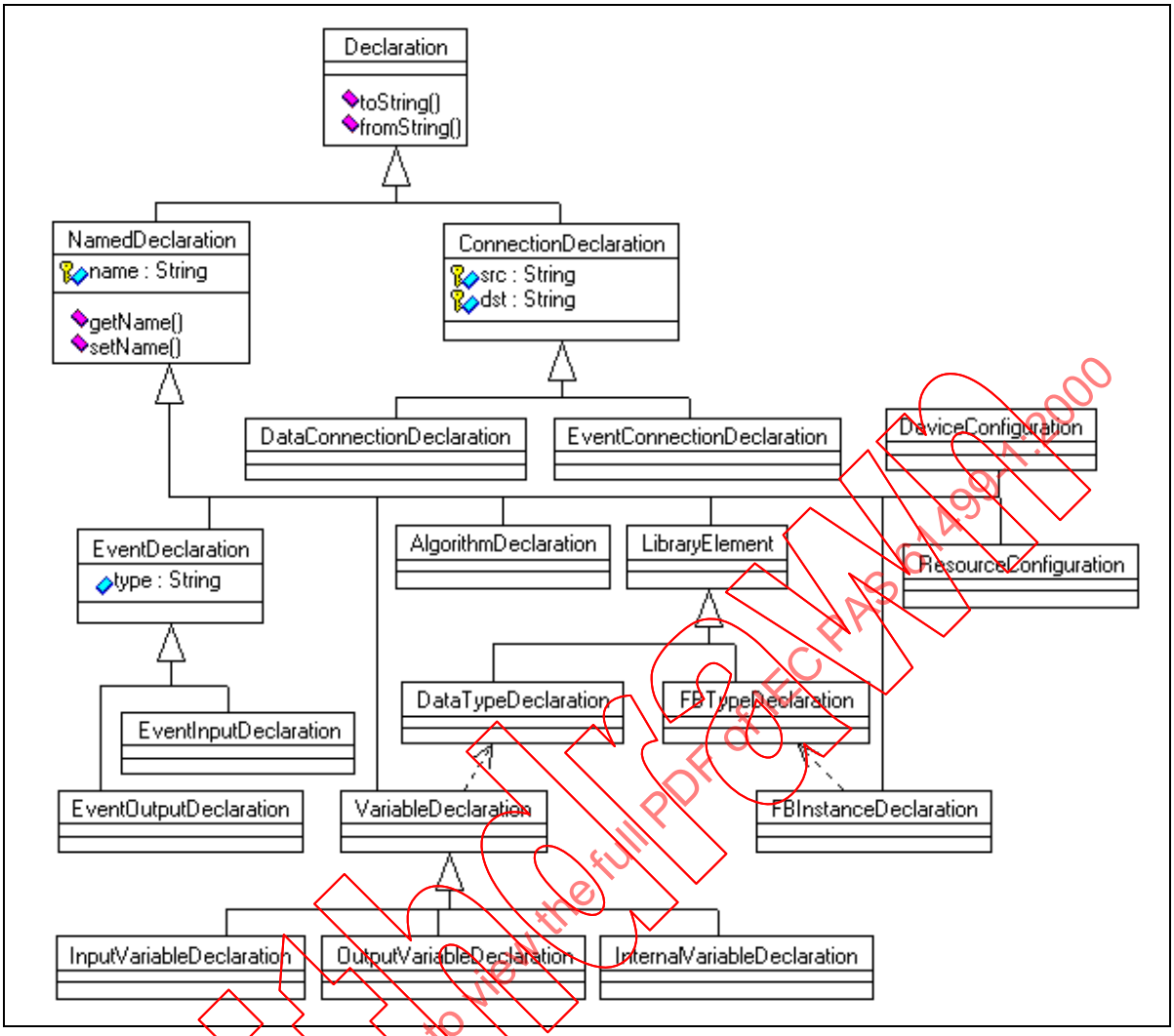


Figure C.1.2 - Declarations

Table C.1.2 - Syntactic productions for declarations

Class	Syntactic production
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`
ResourceConfiguration	`resource_instance`

C.1.3. Function block network declarations

Figure C.1.3 shows the relationships among the elements of *function block network declarations*. See C.1.2 for definitions of the aggregated classes in this diagram.

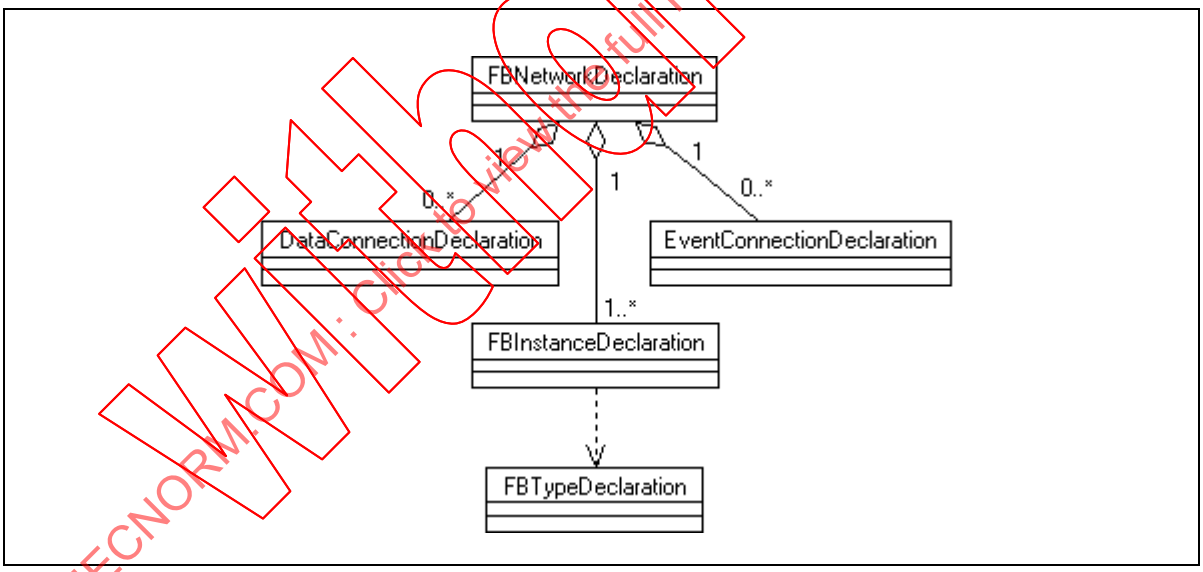


Figure C.1.3 - Function block network declarations

C.1.4. Function block type declarations

Figure C.1.4 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.1.2. The syntactic productions `fb_ecc_declaration` and `fb_service_declaration` in Annex B.2 of IEC 61499-1 correspond to classes **ECCDeclaration** and **ServiceDeclaration**, respectively.

NOTES

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

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

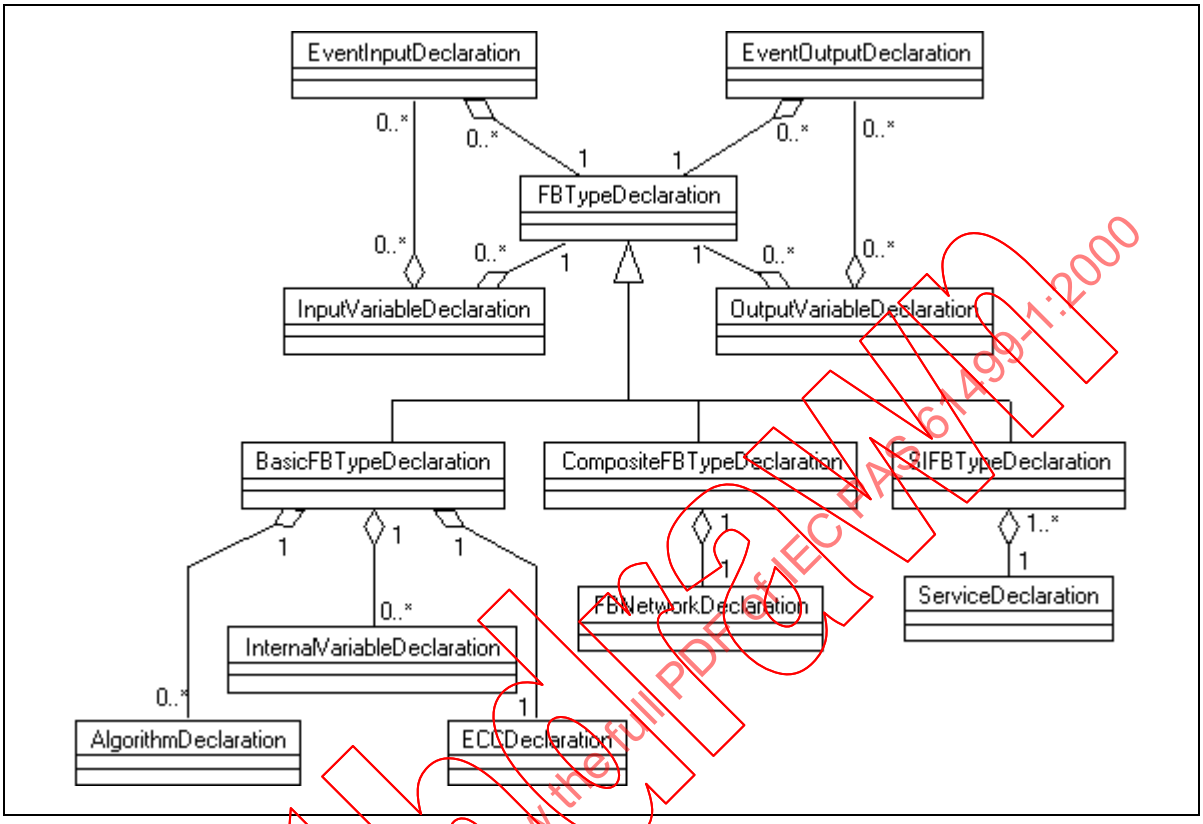


Figure C.1.4 - Function block type declarations

C.2. IPMCS models

Figure C.2-1 presents an overview of the major classes in the industrial-process measurement and control system (IPMCS). Descriptions of the classes in Figure C.2-1 and their corresponding objects in the Engineering Support System (ESS) are given in Table C.2-1.

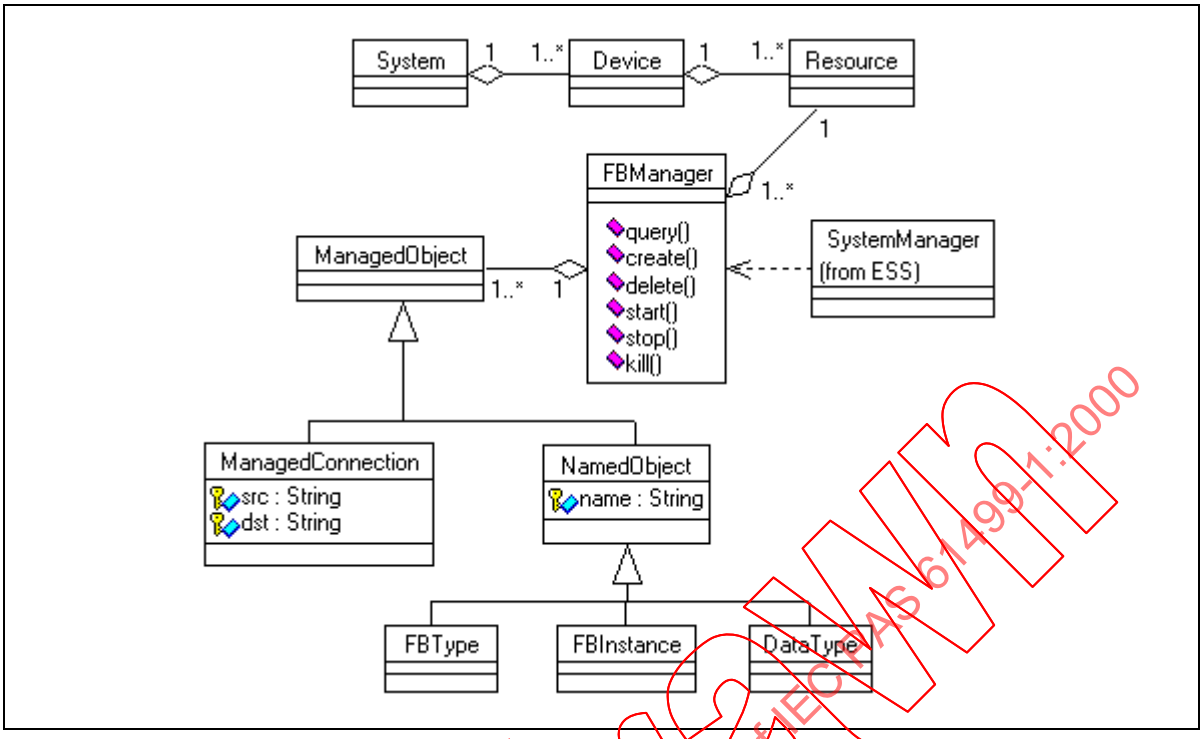


Figure C.2-1 - IPMCS overview

Table C.2-1 - IPMCS classes

IPMCS class	Description	Corresponding ESS class
DataType	An instance of this class is a *data type* as defined in IEC 61499-1.	**DataTypeDeclaration**
Device	An instance of this class represents a *device* as defined in IEC 61499-1.	**DeviceConfiguration**
FBInstance	An instance of this class is a *function block instance* as defined in IEC 61499-1.	**FBInstanceDeclaration**
FBManager	An instance of this class provides the management services defined in IEC 61499-1-3.3.	**SystemManager**
FBType	An instance of this class is a *function block type* as defined in IEC 61499-1.	**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.	**NamedDeclarationt**
Resource	An instance of this class represents a *resource* as defined in IEC 61499-1.	**ResourceConfiguration**
System	An instance of this class represents an Industrial-Process Measurement and Control *System* (IPMCS).	**SystemConfiguration**

Figure C.2-2 shows the relationships among the elements of a *function block instance* and its associated *function block type*. Instances of the classes in Figure C.2-2 represent the identically named objects in IEC 61499-1.

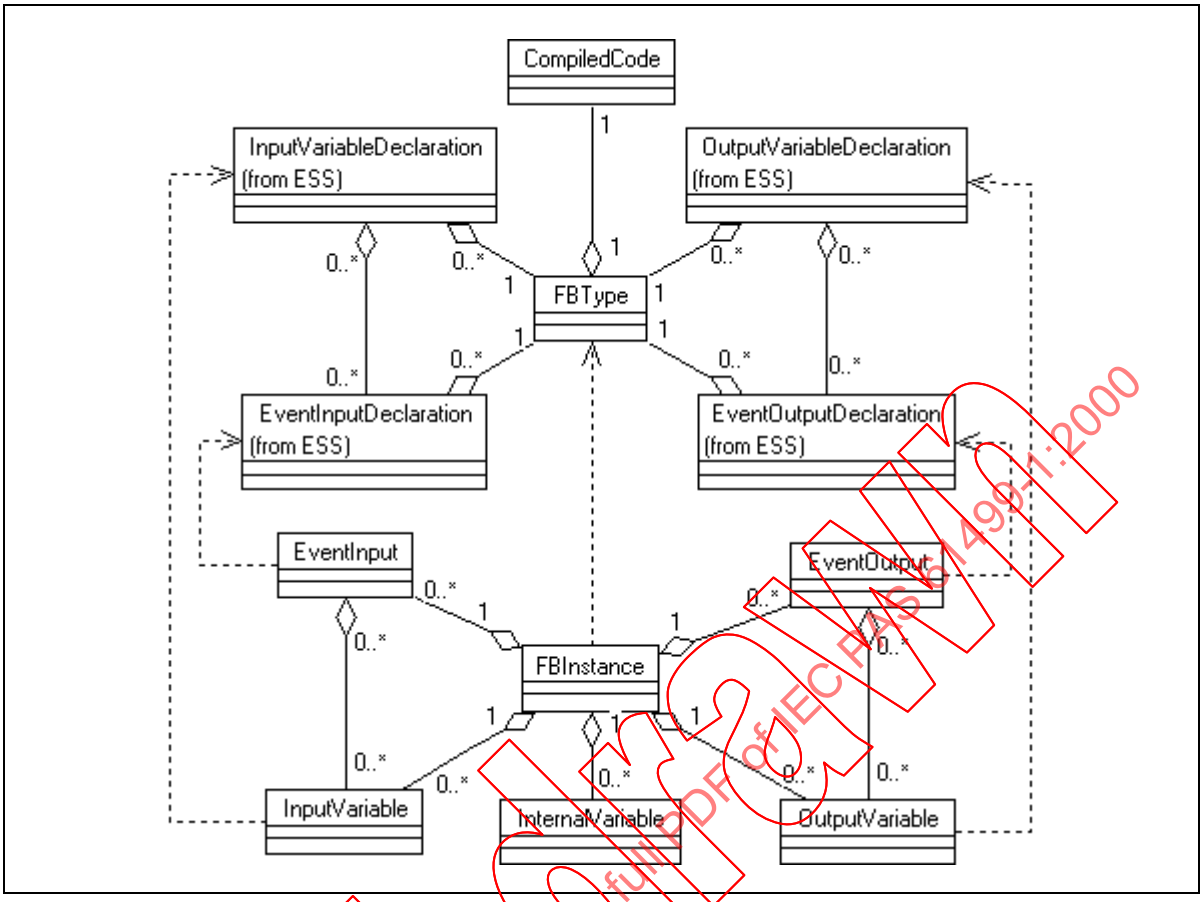


Figure C.2-2 - Function block types and instances

ANNEX D - RELATIONSHIP TO IEC 61131-3(informative)

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 clause 2 of this Part. Annex D.1 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.* Annex D.2 defines event-driven versions of IEC 61131-3 functions and function blocks for the same uses.

D.1. 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:

1. Simple function blocks are represented as *service interface function blocks* for application-initiated interactions as shown in figure 3.1.1(a) of this Part.
2. 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).
3. 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.
4. 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*.

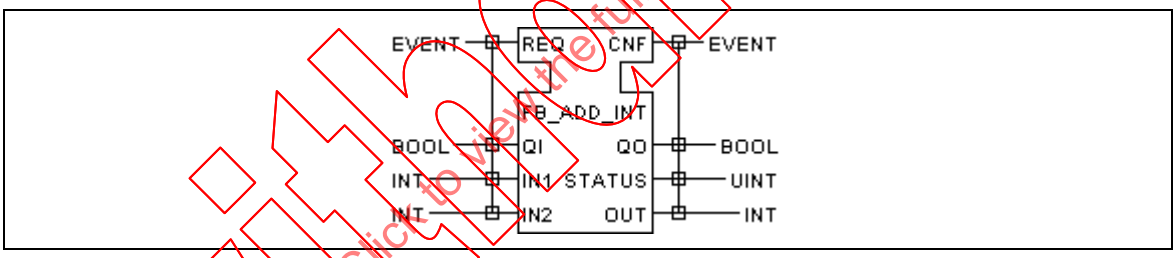


Figure D.1 - Example of a simple function block type

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

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

1. Initialization is as specified in subclause 2.4.2 of IEC 61131-3 for *variables*, and as specified in subclause 2.6 of IEC 61131-3 for Sequential Function Chart (SFC) elements.
2. The occurrence of an INIT+ service primitive is equivalent to "cold restart" initialization as defined in the above mentioned subclauses of IEC 61131-3, followed by an INITO+ service primitive with a STATUS value of zero (0).
3. 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).
4. 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.
5. Successful execution of the algorithm in response to a REQ+ primitive results in a CNF+ primitive with a STATUS value of zero (0).
6. 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

D.2. 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 subclause D.1 with the following modifications:

1. 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.

2. 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 must 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.

ANNEX E - COMMON ELEMENTS (normative)

E.1 Compliance requirement

Implementations of this Specification shall comply with the requirements of subclauses 1.5.1, 2.1, 2.2, 2.3 and 2.4 and the associated elements of Annex B of IEC 61131-3, 2nd edition, for the syntax and semantics of textual representation of common elements, with the exceptions and extensions noted below.

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

E.2. Exceptions

Implementations of this Specification shall **not** utilize the *directly represented variable* notation defined in subclause 2.4.1.1 of IEC 61131-3 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.

E.3 Extensions

The single-subscript notation for specification of array sizes is allowed in implementations of this Specification in addition to the subrange notation defined in IEC 61131-3, e.g., in Feature 6 of Table 18 of IEC 61131-3.

EXAMPLE - The following two declarations are equivalent:

```
VAR
 BITS: ARRAY[0..7] OF BOOL := [1,1,0,0,0,1,0,0];
 TBT: ARRAY[0..1,0..2] OF INT := [1,2,3(4),6];
END_VAR
VAR
 BITS: BOOL[8] := [1,1,0,0,0,1,0,0];
 TBT: INT[2,3] := [1,2,3(4),6];
END_VAR
```

ANNEX F - INFORMATION EXCHANGE (informative)

NOTE - The contents of this Annex may be considered normative in that other Standards may specify that compliance to its provisions is required, and industrial-process measurement and control *systems* and *devices* that comply with its provisions may claim such compliance.

F.1. Use of application layer facilities

NOTE - See ISO/IEC 7498-1 for definitions of terms used in this subclause but not defined in this Specification.

Subclause 7.1.3.2 of ISO/IEC 7498-1 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.* These facilities are provided by the *communication mapping function* of *resources.* This subclause discusses the ways in which communication function blocks may use these facilities, when provided by appropriate application-entities.

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

NOTE 2 - Many of the facilities listed below are not provided by application-entities of industrial-process measurement and control systems (IPMCSs). In this case, the communication function blocks must implement equivalent facilities to provide the required services.

NOTE 3 - 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, this Specification defines transfer syntaxes for both information transfer and application management in Annex F.3.

0) information transfer,
c) synchronization of cooperating applications

Communication function blocks utilize the information transfer facilities provided by application-entities to provide the synchronization represented by the REQ, CNF, IND, and RSP events and to transfer the data represented by the SD inputs and RD outputs.

a) identification of the intended communications partners,
b) determination of the acceptable quality of service,
d) agreement on responsibility for error recovery,
e) agreement on security aspects
g) identification of abstract syntax

These facilities may be used during service initialization as represented by the INIT and INITO events, using elements of the PARAMS data structure as necessary.

f) selection of mode of dialog

These facilities may be used by the specific function block types , e.g., by a SUBSCRIBER to assure that it is interacting properly with a PUBLISHER.

F.2. Communication function block types

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:

- The implementation shall specify the data types and semantics of values of the data inputs and data outputs of each such function block type.
- The implementation shall specify the treatment of abnormal data transfer.
- The implementation shall specify any differences between the behavior of instances of such function block types and the behaviors specified in this clause.

F.2.1. Function blocks for unidirectional transactions

Figures F.2.1-1 through F.2.1-4 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 - See Annex H for full textual specifications of these function block types.

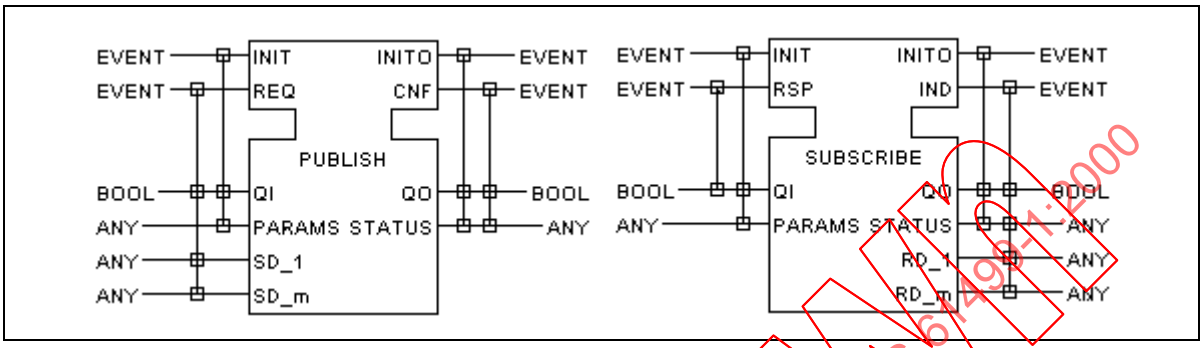


Figure F.2.1-1 - Type specifications for unidirectional transactions

NOTE 1 - The data types and semantics of the PARAMS input and STATUS output are **implementation-dependent**.

NOTE 2 - 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.

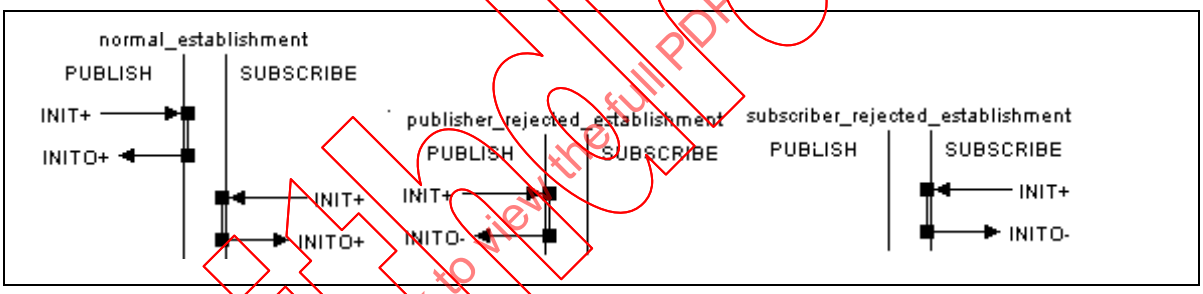


Figure F.2.1-2- Connection establishment for unidirectional transactions

NOTE 1 - The means by which communication connections are set up are beyond the scope of this Specification.

NOTE 2 - Data transfer may be required in order to determine whether the required constraints on RD_1, . . . , RD_m are met per Note 2 of figure F.2.1-1.

NOTE 3 - The transfer syntaxes defined in subclause F.3 may be used to make the determination described above.

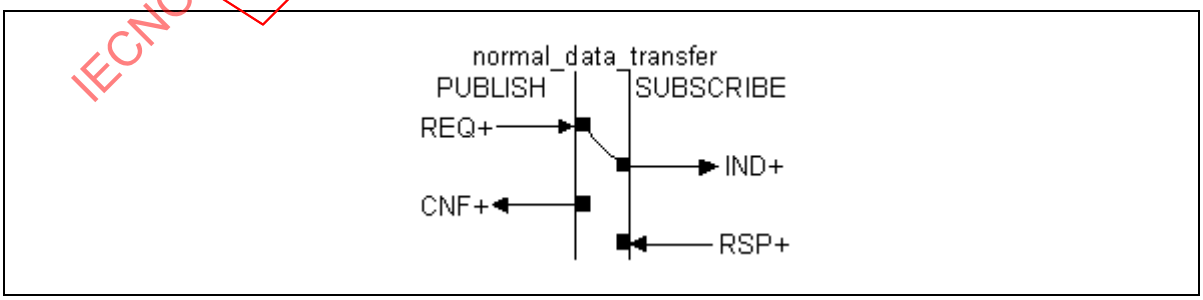


Figure F.2.1-3 - Normal unidirectional data transfer

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

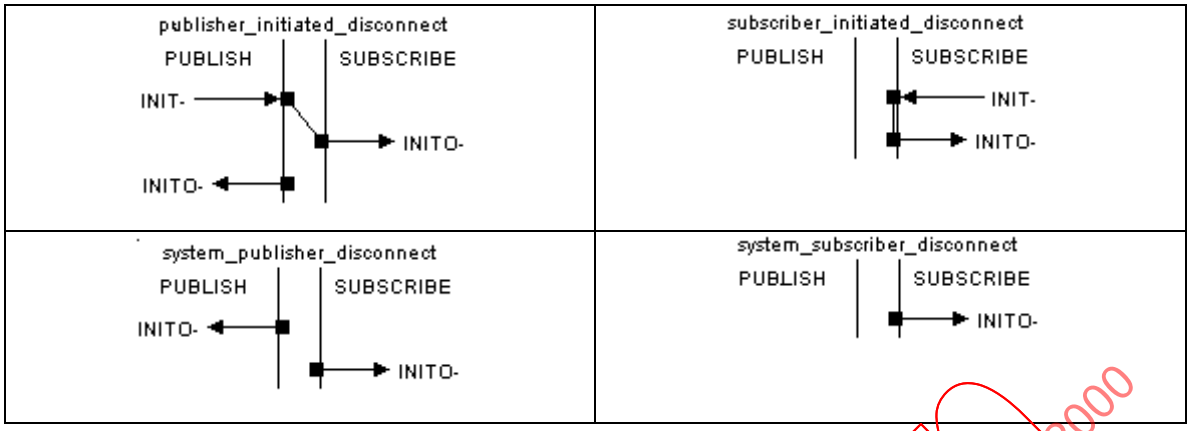


Figure F.2.1-4- Connection release in unidirectional data transfer

F.2.2. Function blocks for bidirectional transactions

Figures F.2.2-1 through F.2.2-4 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 - See Annex H for full textual specifications of these function block types.

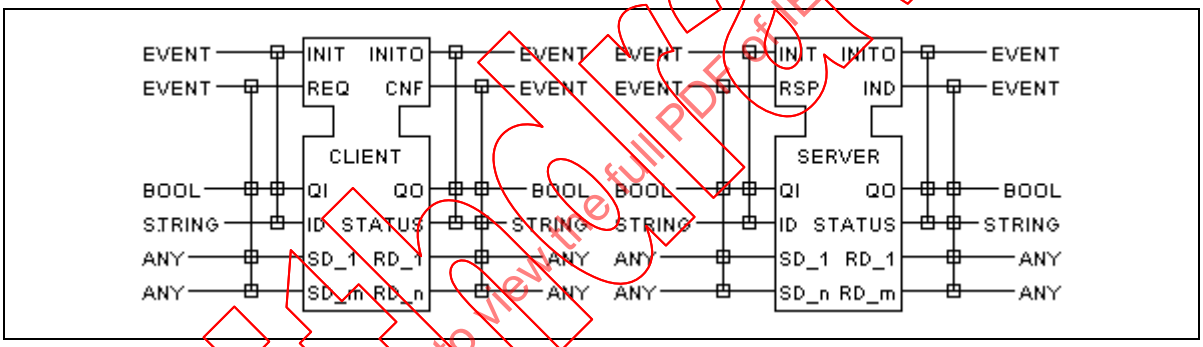


Figure F.2.2-1 - Type specifications for bidirectional transactions

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

2 - 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.

3 - 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.

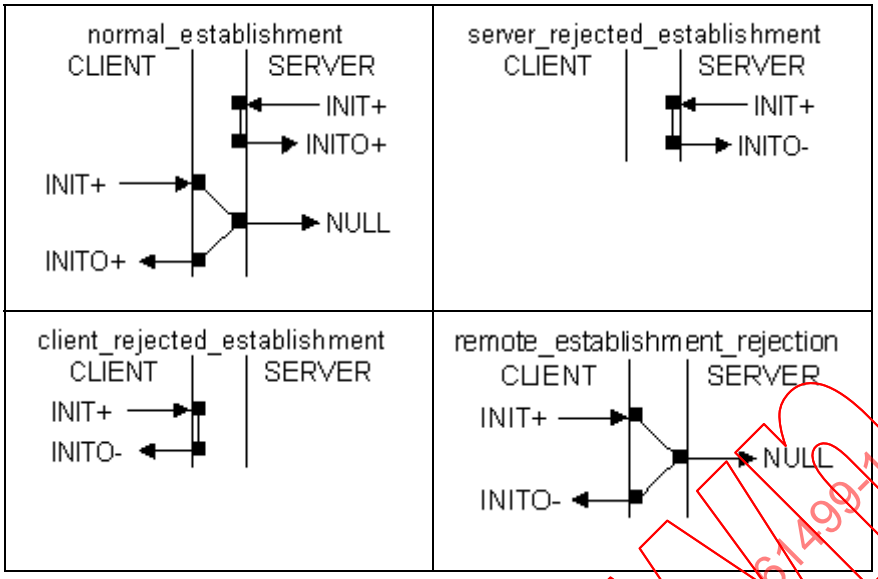


Figure F.2.2-2 - Connection establishment for bidirectional transaction

NOTE 1 - Data transfer may be required in order to determine whether the required constraints on `RD_1, . . . , RD_m` and `RD_1, . . . , RD_n` are met per Notes 2 and 3 of figure F.2.2-1.

NOTE 2 - The transfer syntaxes defined in Annex F.3 may be used to make the determination described above.

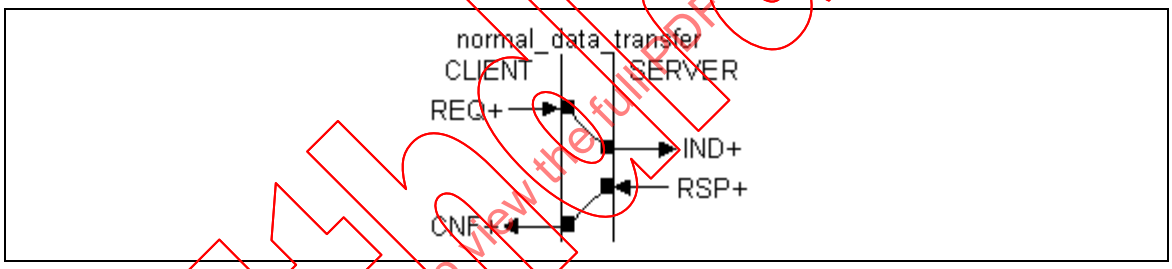


Figure F.2.2-3 - Bidirectional data transfer

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

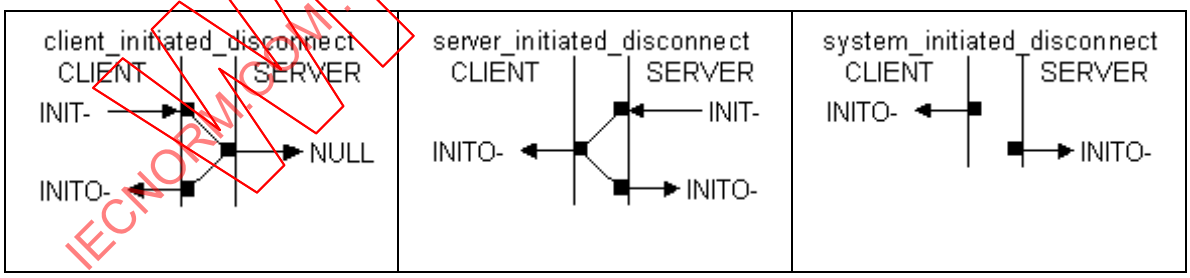


Figure F.2.2-4 - Connection release in bidirectional data transfer
a) Client initiated, b) Server initiated, c) System initiated

F.3. Transfer syntaxes

NOTE 1 - The contents of this subclause may be considered normative in that compliant standards and systems can specify a context within which the specified transfer syntaxes shall be employed.

NOTE 2 - Recommended contexts for the use of these transfer syntaxes are given at the beginning of subclauses F.3.1.1 and F.3.1.2 respectively.

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. This Annex utilizes Abstract Syntax Notation One (ASN.1), as defined in ISO/IEC 8824, to define two abstract syntaxes: IEC61499-FBDATA for normal data transfer and IEC61499-FBMGT for transfer of application management data.

Two sets of encoding rules are given in this Annex:

1, Annex F.3.1 defines BASIC encoding rules.

2. Annex F.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*.

F.3.1. Abstract syntaxes

F.3.1.1. IEC61499-FBDATA

The transfer syntax obtained by applying the COMPACT encoding rules in F.3.2.2 to the abstract syntax in this subclause 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 F.2.1-1 and F.2.2-1.

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

1. 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.
2. 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.
3. The notation `[typeID]` implies that the tag of the data consists of the value of the `typeID` field of the type definition of the corresponding derived data type, established as specified in F.3.1.2.
4. 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 `EnumeratedType`, established as specified in F.3.1.2. The semantics of these data types are as specified for *enumerated data types* in IEC 61131-3.
5. The specific type of a `SubrangeData` item is specified in a `SubrangeType`, established as specified in F.3.1.2. The semantics of these data types are as specified for *subrange data types* in IEC 61131-3.

6. The type of the elements of an ARRAY data item is specified in an `ArrayType`, established as specified in F.3.1.2. The semantics of these data types are as specified for *array data types* in IEC 61131-3.
7. The types of the elements of a STRUCT data item are specified in a `StructuredType` as defined in F.3.1.2. The semantics of these data types are 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 22] 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..4294967296)
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 F.3.2.2.
TIME_OF_DAY::= [APPLICATION 14] IMPLICIT ULINT -- See Table F.3.2.2.
DATE_AND_TIME::= [APPLICATION 15] IMPLICIT ULINT -- See Table F.3.2.2.
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 ::= [typeID] IMPLICIT SEQUENCE OF ElementaryData -- same type
STRUCT ::= [typeID] IMPLICIT SEQUENCE -- different types
END
```

F.3.1.2. IEC61499-FBMGT

The transfer syntax obtained by applying the BASIC encoding described in F.3.2.1 to the abstract syntax in this subclause is recommended for standard encodings of the OBJECT input and RESULT output of instances of the MANAGER function block type defined in subclause 3.3.2.

Table F.3.1.2 shows the recommended usage of the abstract data types defined in this subclause for the encoding of entities declared in the textual syntax specified in Annexes B and E.

Table F.3.1.2 - Use of IEC61499-FBMGT types

IEC61499-FBMGT type	declaration syntax
DataTypeDefinition	type_declaration
FunctionBlockTypeDefinition	fb_type_declaration
FunctionBlockInstanceDefinition	fb_instance_definition
ConnectionDefinition	connection_definition
Identifier	identifier
FunctionBlockInstanceReference	fb_instance_reference
allDataTypes [a]	all_data_types
allFunctionBlockTypes [a]	all_fb_types
ConnectionEndPoint	connection_end_point
AccessPathDefinition	access_path_declaration
AccessPathData	access_path_data
FBData	accessed_data

[a] This is a context-tagged NULL type.

The abstract syntax defined in ASN.1 MODULE IEC61499-FBMGT DEFINITIONS given below is subject to the following semantic RULES:

1. For derived types, the base type and initial value of each element is the same as the type and value of the corresponding "initialValue" element.
2. The rules for initialization of arrays referenced in Annex E, including the rules for incomplete sets of initial values, apply to the use of ArrayType initial values as well as values of ARRAY variables.

ASN.1 MODULE

```
IEC61499-FBMGT DEFINITIONS ::=
BEGIN
IMPORTS FBData, EnumeratedData, FixedLengthInteger FROM IEC61499-FBDATA;
FBCommandObject ::= CHOICE{
     dataTypeDefinition [0] IMPLICIT DataTypeDefinition,
     functionBlockTypeDefinition [1]
        IMPLICIT FunctionBlockTypeDefinition,
     functionBlockInstanceDefinition [2]
        IMPLICIT FunctionBlockInstanceDefinition,
     connectionDefinition [3] IMPLICIT ConnectionDefinition,
     dataTypeName [4] IMPLICIT Identifier,
     functionBlockTypeName [5] IMPLICIT Identifier,
     functionBlockInstanceReference [6]
        IMPLICIT FunctionBlockInstanceReference,
     applicationName [7] IMPLICIT Identifier,
     allDataTypes [8] IMPLICIT NULL,
     allFunctionBlockTypes [9] IMPLICIT NULL,
     connectionEndPoint [10] IMPLICIT ConnectionEndPoint,
     accessPathDefinition [11] IMPLICIT AccessPathDefinition,
     accessPathName [12] IMPLICIT Identifier,
     accessPathData [13] IMPLICIT AccessPathData}
FBCommandResult ::= CHOICE{
     dataTypeName [0] IMPLICIT Identifier,
     functionBlockTypeName [1] IMPLICIT Identifier,
     functionBlockInstanceReference [2]
       IMPLICIT FunctionBlockInstanceReference,
     connectionDefinition [3] IMPLICIT ConnectionDefinition,
     applicationName [4] IMPLICIT Identifier,
     dataTypeList [5] IMPLICIT SEQUENCE OF Identifier,
     functionBlockTypeList [6] IMPLICIT SEQUENCE OF Identifier,
     dataTypeDefinition [7] IMPLICIT DataTypeDefinition,
     functionBlockTypeDefinition [8]
       IMPLICIT FunctionBlockTypeDefinition,
     functionBlockStatus [9] IMPLICIT FunctionBlockStatus,
     connectionEndPoints [10] IMPLICIT SEQUENCE OF ConnectionEndPoint,
     functionBlockInstanceReferences [11]
       IMPLICIT SEQUENCE OF FunctionBlockInstanceReference,
     accessPathDefinition [12] IMPLICIT AccessPathDefinition,
     accessPathName [13] IMPLICIT Identifier,
     accessedData FBData}
Identifier ::= VisibleString -- Syntax per "identifier" in B.4.1
DataTypeDefinition ::= SET{
     typeName [0] IMPLICIT Identifier,
     typeID [1] IMPLICIT TypeID,
     typeSpec DataTypeSpecification,
     description [2] IMPLICIT VisibleString OPTIONAL}
TypeID ::= OCTET STRING -- ISO 8825 encoded tag value
DataTypeSpecification ::= CHOICE{
     directlyDerivedType FBData,
     [0] IMPLICIT EnumeratedType,
     [1] IMPLICIT SubrangeType,
     [2] IMPLICIT ArrayType,
     [3] IMPLICIT StructuredType
EnumeratedType ::= SEQUENCE{
     values [0] IMPLICIT SEQUENCE OF Identifier,
     initialValue [1] IMPLICIT EnumeratedData}
SubrangeType ::= SEQUENCE{
     minValue FixedLengthInteger,
```

```
     maxValue FixedLengthInteger,
     initialValue FixedLengthInteger} -- All of same type
ArrayType ::= SEQUENCE{
     limits [0] IMPLICIT SEQUENCE OF SubscriptLimits,
     initialValue [1] IMPLICIT ARRAY}
SubscriptLimits ::= SEQUENCE{
     minSubscript FixedLengthInteger,
     maxSubscript FixedLengthInteger}
StructuredType ::= SEQUENCE OF VariableDefinition
VariableDefinition ::= SEQUENCE{
     name Identifier,
     initialValue FBData,
     description VisibleString OPTIONAL}
FunctionBlockTypeDefinition ::= SEQUENCE{
     name Identifier,
     eventInputs SEQUENCE OF EventSpecification,
     eventOutputs SEQUENCE OF EventSpecification,
     dataInputs SEQUENCE OF VariableDefinition,
     dataOutputs SEQUENCE OF VariableDefinition,
     body OCTET STRING OPTIONAL}
EventSpecification ::= SEQUENCE{
     name Identifier,
     withList SEQUENCE OF Identifier}
FunctionBlockInstanceDefinition ::= SEQUENCE{
     applicationName Identifier,
     instanceName Identifier,
     typeName Identifier}
FunctionBlockInstanceReference ::= SEQUENCE{
     applicationName Identifier,
     instanceName Identifier}
FunctionBlockStatus ::= ENUMERATED{
     non-existent (0),
     unrunnable (1),
     idle (2),
     running (3),
     stopped (4),
     stopping (6)}
ConnectionDefinition ::= SEQUENCE{
     source ConnectionStartPoint,
     destinations SEQUENCE OF ConnectionEndPoint}
ConnectionStartPoint ::= CHOICE{
     endPoint [0] IMPLICIT ConnectionEndPoint,
     parameter [1] FBData}
ConnectionEndPoint ::= SEQUENCE{
     applicationName Identifier,
     fbInstanceName Identifier,
     attachmentPoint Identifier}
AccessPathDefinition ::= SEQUENCE{
     accessPathName Identifier,
     accessPath STRING} -- per B.3 syntax
AccessPathData ::= SEQUENCE{
     accessPathName Identifier,
     data FBData}
END
```

F.3.2. Encoding rules

F.3.2.1. BASIC encoding

This encoding shall be the result of applying the basic encoding rules of ISO/IEC 8825 to variables of the types defined in Annex F.3.1.

F.3.2.2. COMPACT encoding

This encoding shall be the result of modifying the rules for BASIC encoding given in F.3.2.1 as follows::

1. "Length octets" shall not be included in the encoding of values of the data types shown in table F.3.2.2.
2. The length (in octets) and encoding of the "contents octets" described in ISO/IEC8825 shall be as defined in table F.3.2.2 for values of the data types shown there.
3. Encoding of variables of `TIME`, `DirectlyDerivedData`, `EnumeratedData`, or `SubrangeData` types shall follow the same encoding rules as the base type.
4. "Type octets" shall not be included in the encoding of individual elements of `ARRAY` or `STRUCT` types, except for the encoding of elements of type `BOOL`.
5. The encoding of values of `STRING` and `WSTRING` types shall be primitive.

Table F.3.2.2 - 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 F.3.2.2

(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 F.3.1.1, 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 F.3.1.1.

(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 IEC 559, where the "lsb" defined in IEC 559 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 8824.

(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 G - DEVICE AND RESOURCE MANAGEMENT (informative)

G.1. Device management

A *device* may provide a *management resource* for the purpose of managing other resources within the device. This resource may contain a *device management application* with the same *instance name* as the device, consisting of:

- a *management function block* providing the device management function;
- a *communication function block* providing a *bidirectional transaction service* for the conveyance of the CMD and OBJECT inputs and STATUS and RESULT outputs of the management function block;
- **implementation-dependent** logic for the establishment and maintenance of the communication and management functions.

A configuration meeting these requirements is illustrated in subclause G.3.

G.2. Resource management

If a *device* contains a *management resource*, the management of *resources* within the device can be achieved according to the following rules:

1) The management resource contains one *management function block* for each of the other *resources* contained in the corresponding *device*.
2) Each resource management function block be interconnected with a *communication function block* as illustrated in subclause G.3.1.
3) Resource management facilities may be included in the *device type* specification for those resources which form part of the device type, or may be created and deleted dynamically through the use of the CREATE and DELETE commands to the device *management function block* as described in subclause 3.3.
4) The state of each resource is "stopped" upon creation or system initialization, i.e., each resource must be started with a START command after its creation or upon system initialization.

A configuration meeting these requirements is described in subclause G.3.2.

If a device does not contain a management resource, then facilities for the management of each resource must be provided by the resource itself. These facilities may be included in the *resource type* specification, or in the *device type* specification containing an instance of the associated resource type.

NOTE - A resource may contain additional management function blocks for local or remote management of its applications.

G.3. Applications of management function blocks

This clause provides examples of *applications* meeting the requirements for *device* and *resource management* given in subclauses G.1 and G.2, respectively.

G.3.1. Device management

Figure G.3.1 illustrates the use of an SERVER *communication function block* providing the required communications for device management as described in subclause G.1.