IEC 61158-6-15

Edition 2.0 2010-08

INTERNATIONAL STANDARD

Industrial communication networks – Fieldbus specifications – Part 6-15: Application layer protocol specification – Type 15 elements

About the IEC

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

About IEC publications

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

- Catalogue of IEC publications: www.iec.ch/searchpub

The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,...). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub

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

- Electropedia: www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv

If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

IEC 61158-6-15

Edition 2.0 2010-08

INTERNATIONAL STANDARD

Industrial communication networks – Fieldbus specifications – Part 6-15: Application layer protocol specification – Type 15 elements

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PRICE CODE XD

ICS 25.04.40; 35.100.70; 35.110

ISBN 978-2-88912-131-1

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

INDUSTRIAL COMMUNICATION NETWORKS – FIELDBUS SPECIFICATIONS –

Part 6-15: Application layer protocol specification – Type 15 elements

FOREWORD

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

NOTE 1 Use of some of the associated protocol types is restricted by their intellectual-property-right holders. In all cases, the commitment to limited release of intellectual-property-rights made by the holders of those rights permits a particular data-link layer protocol type to be used with physical layer and application layer protocols in Type combinations as specified explicitly in the profile parts. Use of the various protocol types in other combinations may require permission from their respective intellectual-property-right holders.

International Standard IEC 61158-6-15 has been prepared by subcommittee 65C: Industrial networks, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2007. This edition constitutes a technical revision.

The main changes with respect to the previous edition are listed below:

- editorial corrections.

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

FDIS	Report on voting
65C/607/FDIS	65C/621/RVD

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

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

A list of all parts of the IEC 61158 series, published under the general title *Industrial communication networks – Fieldbus specifications*, can be found on the IEC web site.

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

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

NOTE 2 The revision of this standard will be synchronized with the other parts of the IEC 61158 series.

INTRODUCTION

This part of IEC 61158 is one of a series produced to facilitate the interconnection of automation system components. It is related to other standards in the set as defined by the “three-layer” fieldbus reference model described in IEC/TR 61158-1.

The application protocol provides the application service by making use of the services available from the data-link or other immediately lower layer. The primary aim of this standard is to provide a set of rules for communication expressed in terms of the procedures to be carried out by peer application entities (AEs) at the time of communication. These rules for communication are intended to provide a sound basis for development in order to serve a variety of purposes:

- as a guide for implementers and designers;
- for use in the testing and procurement of equipment;
- as part of an agreement for the admittance of systems into the open systems environment;
- as a refinement to the understanding of time-critical communications within OSI.

This standard is concerned, in particular, with the communication and interworking of sensors, effectors and other automation devices. By using this standard together with other standards positioned within the OSI or fieldbus reference models, otherwise incompatible systems may work together in any combination.

INDUSTRIAL COMMUNICATION NETWORKS – FIELDBUS SPECIFICATIONS –

Part 6-15: Application layer protocol specification – Type 15 elements

1 Scope

1.1 General

The Fieldbus Application Layer (FAL) provides user programs with a means to access the fieldbus communication environment. In this respect, the FAL can be viewed as a “window between corresponding application programs.”

This standard provides common elements for basic time-critical and non-time-critical messaging communications between application programs in an automation environment and material specific to Type 15 fieldbus. The term “time-critical” is used to represent the presence of a time-window, within which one or more specified actions are required to be completed with some defined level of certainty. Failure to complete specified actions within the time window risks failure of the applications requesting the actions, with attendant risk to equipment, plant and possibly human life.

This standard defines in an abstract way the externally visible behavior provided by the Type 15 fieldbus Application Layer in terms of

a) the abstract syntax defining the application layer protocol data units conveyed between communicating application entities,
b) the transfer syntax defining the application layer protocol data units conveyed between communicating application entities,
c) the application context state machine defining the application service behavior visible between communicating application entities; and
d) the application relationship state machines defining the communication behavior visible between communicating application entities; and.

The purpose of this standard is to define the protocol provided to

a) define the wire-representation of the service primitives defined in IEC 61158-5-15, and
b) define the externally visible behavior associated with their transfer.

This standard specifies the protocol of the Type 15 IEC fieldbus Application Layer, in conformance with the OSI Basic Reference Model (ISO/IEC 7498) and the OSI Application Layer Structure (ISO/IEC 9545).

1.2 Specifications

The principal objective of this standard is to specify the syntax and behavior of the application layer protocol that conveys the application layer services defined in IEC 61158-5-15.

A secondary objective is to provide migration paths from previously-existing industrial communications protocols. It is this latter objective which gives rise to the diversity of protocols standardized in IEC 61158-6.

1.3 Conformance

This standard does not specify individual implementations or products, nor does it constrain the implementations of application layer entities within industrial automation systems. Conformance is achieved through implementation of this application layer protocol specification.

2 Normative references

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

IEC 61158-5-15:2010[1], *Industrial communication networks – Fieldbus specifications - Part 5-15: Application layer service definition – Type 15 elements*

ISO/IEC 7498-1, *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*

ISO/IEC 8822, *Information technology – Open Systems Interconnection – Presentation service definition*

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

ISO/IEC 9545, *Information technology – Open Systems Interconnection – Application Layer structure*

3 Terms and definitions, abbreviations, symbols and conventions

3.1 Terms and definitions

For the purposes of this document, the following terms as defined in these publications apply:

3.1.1 ISO/IEC 7498-1 terms

a) application entity
b) application process
c) application protocol data unit
d) application service element
e) application entity invocation
f) application process invocation
g) application transaction
h) real open system
i) transfer syntax

3.1.2 ISO/IEC 8822 terms

a) abstract syntax
b) presentation context

---

1 To be published.

3.1.3 ISO/IEC 9545 terms

a) application-association
b) application-context
c) application context name
d) application-entity-invocation
e) application-entity-type
f) application-process-invocation
g) application-process-type
h) application-service-element
i) application control service element

3.1.4 ISO/IEC 8824-1 terms

a) object identifier
b) type

3.1.5 IEC/TR 61158-1 terms

The following IEC/TR 61158-1 terms apply.

3.1.5.1
application
function or data structure for which data is consumed or produced

3.1.5.2
application layer interoperability
capability of application entities to perform coordinated and cooperative operations using the services of the FAL

3.1.5.3
application object
object class that manages and provides the run time exchange of messages across the network and within the network device

NOTE Multiple types of application object classes may be defined.

3.1.5.4
application process
part of a distributed application on a network, which is located on one device and unambiguously addressed

3.1.5.5
application process identifier
distinguishes multiple application processes used in a device

3.1.5.6
application process object
component of an application process that is identifiable and accessible through an FAL application relationship

NOTE Application process object definitions are composed of a set of values for the attributes of their class.

3.1.5.7
application process object class
class of application process objects defined in terms of the set of their network-accessible attributes and services

3.1.5.8
application relationship
cooperative association between two or more application-entity-invocations for the purpose of exchange of information and coordination of their joint operation

NOTE This relationship is activated either by the exchange of application-protocol-data-units or as a result of preconfiguration activities.

3.1.5.9
application relationship endpoint
context and behavior of an application relationship as seen and maintained by one of the application processes involved in the application relationship

NOTE Each application process involved in the application relationship maintains its own application relationship endpoint.

3.1.5.10
application service element
application-service-element that provides the exclusive means for establishing and terminating all application relationships

3.1.5.11
attribute
description of an externally visible characteristic or feature of an object

NOTE The attributes of an object contain information about variable portions of an object. Typically, they provide status information or govern the operation of an object. Attributes may also affect the behavior of an object. Attributes are divided into class attributes and instance attributes.

3.1.5.12
behavior
indication of how the object responds to particular events

NOTE Its description includes the relationship between attribute values and services.

3.1.5.13
class
set of objects, all of which represent the same kind of system component

NOTE A class is a generalization of the object; a template for defining variables and methods. All objects in a class are identical in form and behavior, but usually contain different data in their attributes.

3.1.5.14
class attributes
attribute that is shared by all objects within the same class

3.1.5.15
class code

unique identifier assigned to each object class

3.1.5.16
class specific service
service defined by a particular object class to perform a required function which is not performed by a common service

NOTE A class specific object is unique to the object class which defines it.

3.1.5.17
Client

(a) object which uses the services of another (server) object to perform a task

(b) initiator of a message to which a server reacts, such as the role of an AR endpoint in which it issues confirmed service request APDUs to a single AR endpoint acting as a server

3.1.5.18
conveyance path
unidirectional flow of APDUs across an application relationship

3.1.5.19
cyclic
term used to describe events which repeat in a regular and repetitive manner

3.1.5.20
dedicated AR
AR used directly by the FAL user

NOTE On Dedicated ARs, only the FAL Header and the user data are transferred.

3.1.5.21
device
physical hardware connection to the link

NOTE A device may contain more than one node.

3.1.5.22
device profile
collection of device dependent information and functionality providing consistency between similar devices of the same device type

3.1.5.23
dynamic AR
AR that requires the use of the AR establishment procedures to place it into an established state

3.1.5.24
endpoint
one of the communicating entities involved in a connection

3.1.5.25
error
discrepancy between a computed, observed or measured value or condition and the specified or theoretically correct value or condition

3.1.5.26
error class
general grouping for error definitions

NOTE Error codes for specific errors are defined within an error class.

3.1.5.27
error code
identification of a specific type of error within an error class

3.1.5.28
FAL subnet
networks composed of one or more data link segments

NOTE Subnets are permitted to contain bridges, but not routers. FAL subnets are identified by a subset of the network address.

3.1.5.29
logical device
FAL class that abstracts a software component or a firmware component as an autonomous self-contained facility of an automation device

3.1.5.30
management information
network-accessible information that supports managing the operation of the fieldbus system, including the application layer

NOTE Managing includes functions such as controlling, monitoring, and diagnosing.

3.1.5.31
network
series of nodes connected by some type of communication medium

NOTE The connection paths between any pair of nodes can include repeaters, routers and gateways.

3.1.5.32
peer
role of an AR endpoint in which it is capable of acting as both client and server

3.1.5.33
pre-defined AR endpoint
AR endpoint that is defined locally within a device without use of the create service

NOTE Pre-defined ARs that are not pre-established are established before being used.

3.1.5.34
pre-established AR endpoint
AR endpoint that is placed in an established state during configuration of the AEs that control its endpoints

3.1.5.35
Publisher
role of an AR endpoint in which it transmits APDUs onto the fieldbus for consumption by one or more subscribers

NOTE The publisher may not be aware of the identity or the number of subscribers and it may publish its APDUs using a dedicated AR. Two types of publishers are defined by this standard, Pull Publishers and Push Publishers, each of which is defined separately.

3.1.5.36
server

a) role of an AREP in which it returns a confirmed service response APDU to the client that initiated the request

b) object which provides services to another (client) object

3.1.5.37
service
operation or function than an object and/or object class performs upon request from another object and/or object class

NOTE A set of common services is defined and provisions for the definition of object-specific services are provided. Object-specific services are those which are defined by a particular object class to perform a required function which is not performed by a common service.

3.1.5.38
subscriber
role of an AREP in which it receives APDUs produced by a publisher

NOTE Two types of subscribers are defined by this standard, pull subscribers and push subscribers, each of which is defined separately.

3.1.6 Specific definitions for client/server

3.1.6.1
coils, discrete outputs
application process object, a set of coils, characterized by the address of a coil and a quantity of coils, this set is also called discrete outputs when associated with field outputs

3.1.6.2
discrete, discrete input
application process object, addressed by an unsigned number and having a width of one bit, representing a 1-bit encoded status value, read-only, with the value '1' encoding the status ON and the value '0' encoding the status OFF, also called discrete input, especially when associated with field inputs

3.1.6.3
discrete inputs, discretes
application process object, a set of discretes, characterized by the address of a discrete and a quantity of discretes, this set is also called discrete inputs, especially when associated with field inputs

3.1.6.4
coil, discrete output
application process object, addressed by an unsigned number and having a width of one bit, representing a 1-bit encoded status value, read-write, with the value '1' encoding the status ON and the value '0' encoding the status OFF, also called discrete output when associated with field output

3.1.6.5
encapsulated interface
mechanism encapsulating a service for an interface, which is an application process object characterized by an MEI type

3.1.6.6
exception
encoding used to signal a service request failure

3.1.6.7
exception code
encoding associated with an exception, detailing the reason of a service request failure

3.1.6.8
file
application process object, an organization of records, characterized by an unsigned number

3.1.6.9
function code
encoding of a service requested to a server

3.1.6.10
holding register, output register
application process object, addressed by an unsigned number and representing values with 16 bits, read-write, also called output register, especially when associated with field outputs

3.1.6.11
holding registers, output registers
application process object, a set of holding registers, characterized by the address of a holding register and a quantity of holding registers, also called output registers, especially when associated with field outputs

3.1.6.12
input register
application process object, addressed by an unsigned number and representing values with 16 bits, read-only

3.1.6.13
input registers
application process object, a set of input registers, characterized by the address of an input register and a quantity of input registers

3.1.6.14
record
application process object, a set of contiguous registers of a specified type, characterized by the address of the first register and by the quantity of registers; in the context of this definition, the registers involved have also been called references

3.1.6.15
reference
denigrated term for register

3.1.6.16
reference type
denigrated term for register type

3.1.6.17
sub-code
specialization of a function code

3.1.6.18
unit ID
logical device identifier

3.1.6.19
MEI type
type specified as an octet value, used to dispatch a service to the appropriate interface in the context of the encapsulated interface mechanism

3.1.7 Specific definitions for publish/subscribe

3.1.7.1
network object
Publish/subscribe application, reader, or writer

3.1.7.2
GUID
globally unique network object identifier

3.1.7.3
composite state
attributes of a set of network objects

3.1.7.4
reader
subscriber or a CSTReader

3.1.7.5
writer
Publisher or a CSTWriter

3.1.7.6
CSTReader
meta-information-specialized subscriber

3.1.7.7
CSTWriter
meta-information-specialized publisher

3.1.7.8
communication actor
reader or writer

3.1.7.9
domain participant
application that contains publish/subscribe elements, also called publish/subscribe application

NOTE This terminology is adopted to avoid the overuse of the term “application”. At the same time, the term “domain” has a place within publish/subscribe. The Type extensibility allows for the concept of “domains”, or independent communication planes, effectively permitting isolation of application exchanges within domains. While OMG DDS as in “Data Distribution Service for Real-Time Systems Specification, Version 1.1, December 2005” uses this extension, the feature will not be examined further in this specification, which will consider a single domain.

3.1.7.10
manager
specialized publish/subscribe application, containing specialized publishers and subscribers, and involved in the described discovery and maintenance mechanism; not to be confused with any publishing manager

3.1.7.11
managed participant
a publish/subscribe application; the qualifier refers to its role in relation to a manager when involved in the described discovery and maintenance mechanism; not to be confused with any publishing manager subordinate

3.1.7.12
sequence number
number used to uniquely identify elementary publish/subscribe messages in an ordered manner

3.2 Abbreviations and symbols

3.2.1 Common abbreviations and symbols

AE	Application Entity
AL	Application Layer
ALME	Application Layer Management Entity
ALP	Application Layer Protocol
APO	Application Object
AP	Application Process
APDU	Application Protocol Data Unit

API	Application Process Identifier
AR	Application Relationship
AREP	Application Relationship End Point
ASCII	American Standard Code for Information Interchange
ASE	Application Service Element
Cnf	Confirmation
DL-	(as a prefix) data-link-
DLC	data-link Connection
DLCEP	data-link Connection End Point
DLL	data-link Layer
DLM	data-link-management
DLSAP	data-link Service Access Point
DLSDU	DL-service-data-unit
FAL	Fieldbus Application Layer
ID	Identifier
IEC	International Electrotechnical Commission
Ind	Indication
LME	Layer Management Entity
lsb	Least Significant Bit
msb	Most Significant Bit
OSI	Open Systems Interconnect
QoS	Quality of Service
Req	Request
Rsp	Response
SAP	Service Access Point
SDU	Service Data Unit
SMIB	System Management Information Base
SMK	System Management Kernel

3.2.2 Abbreviations and symbols for client/server

C/S	Client/Server
FC	Function Code
CAN	Controller Area Network
CiA	CAN in Automation
MEI	Encapsulated Interface type
URL	Uniform Resource Locator

3.2.3 Abbreviations and symbols for publish/subscribe

CS	Composite State
DCPS	Data-Centric Publish-Subscribe
DDS	Data Disribution Service
DLRL	Data Local Reconstruction Layer
OMG	Object Management Group
P/S	Publish/Subscribe

3.3 Conventions

3.3.1 Overview

The FAL is defined as a set of object-oriented ASEs. Each ASE is specified in a separate subclause. Each ASE specification is composed of two parts, its class specification, and its service specification.

The class specification defines the attributes of the class. The attributes are accessible from instances of the class using the Object Management ASE services specified in Clause 5 of this standard. The service specification defines the services that are provided by the ASE.

3.3.2 General conventions

This standard uses the descriptive conventions given in ISO/IEC 10731

3.3.3 Conventions for class definitions

Class definitions are described using templates. Each template consists of a list of attributes for the class. The general form of the template is shown below:

FAL ASE: ASE Name
CLASS:Class Name
CLASS ID: #
PARENT CLASS: Parent Class Name
ATTRIBUTES:

1	(o)	Key Attribute:	numeric identifier
2	(o)	Key Attribute:	name
3	(m)	Attribute:	attribute name(values)
4	(m)	Attribute:	attribute name(values)
4.1	(s)	Attribute:	attribute name(values)
4.2	(s)	Attribute:	attribute name(values)
4.3	(s)	Attribute:	attribute name(values)
5.	(c)	Constraint:	constraint expression
5.1	(m)	Attribute:	attribute name(values)
5.2	(o)	Attribute:	attribute name(values)
6	(m)	Attribute:	attribute name(values)
6.1	(s)	Attribute:	attribute name(values)
6.2	(s)	Attribute:	attribute name(values)

SERVICES:

1	(o)	OpsService:	service name
2.	(c)	Constraint:	constraint expression
2.1	(o)	OpsService:	service name
3	(m)	MgtService:	service name

(1) The "FAL ASE:" entry is the name of the FAL ASE that provides the services for the class being specified.

(2) The "CLASS:" entry is the name of the class being specified. All objects defined using this template will be an instance of this class. The class may be specified by this standard, or by a user of this standard.

(3) The "CLASS ID:" entry is a number that identifies the class being specified. This number is unique within the FAL ASE that will provide the services for this class. When qualified by the identity of its FAL ASE, it unambiguously identifies the class within the scope of the FAL. The value "NULL" indicates that the class cannot be instantiated. Class IDs between 1 and 255 are reserved by this standard to identify standardized classes. They have been assigned to maintain compatibility with existing national standards. CLASS IDs between 256 and 2 048 are allocated for identifying user defined classes.

(4) The "PARENT CLASS:" entry is the name of the parent class for the class being specified. All attributes defined for the parent class and inherited by it are inherited for

the class being defined, and therefore do not have to be redefined in the template for this class.

NOTE The parent-class "TOP" indicates that the class being defined is an initial class definition. The parent class TOP is used as a starting point from which all other classes are defined. The use of TOP is reserved for classes defined by this standard.

(5) The "ATTRIBUTES" label indicate that the following entries are attributes defined for the class.

a) Each of the attribute entries contains a line number in column 1, a mandatory (m) / optional (o) / conditional (c) / selector (s) indicator in column 2, an attribute type label in column 3, a name or a conditional expression in column 4, and optionally a list of enumerated values in column 5. In the column following the list of values, the default value for the attribute may be specified.

b) Objects are normally identified by a numeric identifier or by an object name, or by both. In the class templates, these key attributes are defined under the key attribute.

c) The line number defines the sequence and the level of nesting of the line. Each nesting level is identified by period. Nesting is used to specify

i) fields of a structured attribute (4.1, 4.2, 4.3),

ii) attributes conditional on a constraint statement (5). Attributes may be mandatory (5.1) or optional (5.2) if the constraint is true. Not all optional attributes require constraint statements as does the attribute defined in (5.2).

iii) the selection fields of a choice type attribute (6.1 and 6.2).

(6) The "SERVICES" label indicates that the following entries are services defined for the class.

a) An (m) in column 2 indicates that the service is mandatory for the class, while an (o) indicates that it is optional. A (c) in this column indicates that the service is conditional. When all services defined for a class are defined as optional, at least one has to be selected when an instance of the class is defined.

b) The label "OpsService" designates an operational service (1).

c) The label "MgtService" designates an management service (2).

d) The line number defines the sequence and the level of nesting of the line. Each nesting level is identified by period. Nesting within the list of services is used to specify services conditional on a constraint statement.

3.3.4 Conventions for service definitions

3.3.4.1 General

The FAL is defined as a set of object-oriented ASEs. Each ASE is specified in a separate subclause. Each ASE specification is composed of three parts: its class definitions, its services, and its protocol specification. The first two are contained in IEC 61158-5-15. The protocol specification for each of the ASEs is defined in this standard.

The class definitions define the attributes of the classes supported by each ASE. The attributes are accessible from instances of the class using the Management ASE services specified in IEC 61158-5-15. The service specification defines the services that are provided by the ASE.

This standard uses the descriptive conventions given in ISO/IEC 10731.

3.3.5 Conventions for class definitions

The data-link layer mapping definitions are described using templates. Each template consists of a list of attributes for the class. The general form of the template is defined in IEC/TR 61158-1.

3.3.6 Abstract syntax conventions

When the "optionalParametersMap" parameter is used, a bit number which corresponds to each OPTIONAL or DEFAULT production is given as a comment.

3.4 Conventions used in state machines

The state machines are described in Table 1:

Table 1 – Conventions used for state machines

#	Current state	Event / condition => action	Next state
Name of this transition.	The current state to which this state transition applies.	Events or conditions that trigger this state transaction. => The actions that are taken when the above events or conditions are met. The actions are always indented below events or conditions.	The next state after the actions in this transition is taken.

The conventions used in the state machines are as follows:

:= Value of an item on the left is replaced by value of an item on the right. If an item on the right is a parameter, it comes from the primitive shown as an input event.

xxx A parameter name.

EXAMPLE 1
Identifier := reason
means value of a 'reason' parameter is assigned to a parameter called 'Identifier.'

"xxx" Indicates fixed value.

EXAMPLE 2
Identifier := "abc"
means value "abc" is assigned to a parameter named 'Identifier.'

= A logical condition to indicate an item on the left is equal to an item on the right.

< A logical condition to indicate an item on the left is less than the item on the right.

> A logical condition to indicate an item on the left is greater than the item on the right.

<> A logical condition to indicate an item on the left is not equal to an item on the right.

&& Logical "AND"

|| Logical "OR"

This construct allows the execution of a sequence of actions in a loop within one transition. The loop is executed for all values from start_value to end_value.

EXAMPLE 3
for (Identifier := start_value to end_value)
actions
endfor

This construct allows the execution of alternative actions depending on some condition (which might be the value of some identifier or the outcome of a previous action) within one transition.

```
EXAMPLE 4
   If (condition)
      actions
   else
      actions
   endif
```

Readers are strongly recommended to refer to the subclauses for the AREP attribute definitions, the local functions, and the FAL-PDU definitions to understand protocol machines. It is assumed that readers have sufficient knowledge of these definitions, and they are used without further explanations.

4 Abstract syntax for client/server

The abstract syntax of APDUs is combined with their transfer syntax and is specified in Clause 5.

5 Transfer syntax for client/server

5.1 General

The sending Application Layer prepares an APDU to transfer to the receiving Application Layer. It uses the parameters from the service primitives to do so. There are several formats of the APDU:

— request APDU from Client to Server device or devices, or

— normal response and positive confirmation from Server to Client device, or

— response and negative confirmation from Server to Client device.

The format and coding rules for all APDUs are specified in this clause.

5.2 Common APDU structure

All APDUs have a common structure as shown in Figure 1.

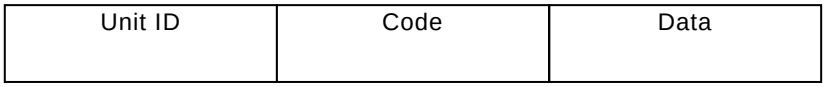

Figure 1 – APDU Format

5.2.1 Unit ID

Client/server devices in the role of servers are addressed using a Unit ID. The Unit ID needs to be unique across all the servers addressable by a client. The set of addressable servers is determined by the underlying layer. This set is sometimes called connection.

The Unit ID assignment is outside of the scope of this specification.

The Unit ID identifies logical devices. There may be more than one logical device per physical device.

Some values of Unit ID are reserved and have particular meanings. The value 0 is reserved for broadcast.

Field Type: Unsigned8

Allowed values: 1 to 247, and 0 for broadcast where supported.

NOTE In general the Unit ID is only required for logical devices having the role of servers. Often logical devices can have either role or multiple roles, via configuration, and their Unit ID is not used when they only perform in the client role. Depending on the underlying layers some devices can have concurrently a client and a server role on the same access point, this is the case for client/server on Token Bus/HDLC, outside the scope of this specification.

5.2.2 Code

5.2.2.1 General

This field represents:

— a requested service, confirmed or unconfirmed, via an identifier called function code, or

— the normal response and positive confirmation of a requested service, represented by echoing the requested service identifier, or

— the exception response and negative confirmation of a requested service, represented by echoing the requested service identifier with its high bit turned on. The latter representation is also called exception.

Field Type: Unsigned8

5.2.2.2 Service identifiers and function codes

Client/server service identifiers are commonly called function codes.

Function codes are encodings of services requested to a server. Some function codes are further specialized by means of a sub-code, specified as part of the data field. These encodings are partitioned in three categories, and since the subdivision may propagate to the sub-codes, for sake of completeness, despite being part of the data field they will also be mentioned here:

Publicly assigned function codes

These function codes are either assigned to a standard service or reserved for future assignment. The standard services and their identifiers will be detailed in this specification.

User definable function codes

These function codes can be used for experimentation in a controlled laboratory environment. They must not be used in an open environment.

Ranges: There are two ranges, FC 65 (0x41) to 72 (0x48) included, and 100 (0x64) to 110 (0x6E) included.

Reserved function codes

These function codes are currently used by some companies for legacy products and are not available for public use.

NOTE 1 Function code assignments are managed by the Modbus-IDA industrial consortium.

NOTE 2 The following function codes, while publicly assigned, are not covered by this specification: FC 7 (0x07, Read Exception Status), FC 8 (0x08, Diagnostics), FC 11 (0x0B, Get Com Event Counter), FC 12 (0x0C, Get Com Event Log), FC 17 (0x11, Report Slave ID).

NOTE 3 The following function codes and function code/sub-codes are reserved: FC 8/19 (0x08/0x13), FC 8/21-255 (0x08/0x15-0xFFFF), FC 9 (0x09), FC 10 (0x0A), FC 13 (0x0D), FC 14 (0x0E), FC 41 (0x29), FC 42 (0x2A), FC 43/0-12 (0x2B/0x00-0x0C), FC 43/15-255 (0x2B/0x0F-0xFF), FC 90 (0x5A), FC 91 (0x5B), FC 125 (0x7D), FC 126 (0x7E), FC 127 (0x7F).

5.2.3 Data

For normal requests and responses this is the user data which is transferred between the application layer and its user. The application layer assembles it from the parameters of a service primitive or parses it into parameters of a service primitive. Its structure depends upon the type of APDU. For exception responses it represents the reason for the exception via an exception code.

Field Type: from 1 to 252 octets; the Type is APDU specific.

5.2.4 Client to server confirmed service request

The format is as in Figure 2.

Unit ID	Function code	Request Data

Figure 2 – Client to server confirmed service request

5.2.5 Normal response from server to client

The format for the normal response to a confirmed service is as in Figure 3. The Unit ID and the function code fields are the same as the corresponding fields in the request.

Unit ID	Function code	Response Data

Figure 3 – Normal response from server to client

5.2.6 Exception response from server to client

The format for the exception response is as in Figure 4. The Unit ID is the same as the corresponding field in the request. The exception is produced adding 0x80 to the function code of the corresponding request.

Unit ID	Exception	Exception code

Figure 4 – Exception response from server to client

The exception codes, giving information on the service failure, are shown in Table 2.

Table 2 – Exception code

Encoding	Name	Description
0x01	Illegal function	The function code received in the query is not an allowable action for the server. This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server is in the wrong state to process a request of this type, for example because it is not configured and it is being asked to return register values
0x02	Illegal data address	The data address received in the query is not an allowable address for the server. More specifically, the combination of reference number and transfer length is invalid. Example For a controller with 100 registers, a request with offset (data address) 96 and length 4 would succeed, a request with offset 96 and length 5 would generate exception code 0x02
0x03	Illegal data value	A value contained in the query data field is not an allowable value for server. This indicates a fault in the structure of the remainder of a complex request, such as that the implied length is incorrect. It specifically does NOT mean for example that a data item submitted for storage in a register has a value outside the expectation of the application program, since the client/server protocol is unaware of the significance of any particular value for any particular register
0x04	Server device failure	An unrecoverable error occurred while the server was attempting to perform the requested action
0x05	Acknowledge	The server accepted the service invocation but the service requires a relatively long time to execute. The server therefore returns only an acknowledgement of the service invocation receipt. This response is returned to prevent a timeout error from occurring in the client
0x06	Server busy	The server was unable to accept the request. The client application has the responsibility of deciding if and when to re-send the request
0x08	Memory parity error	For specialized use in conjunction with function codes 20 (0x14) and 21 (0x15), to indicate that the extended file area failed to pass a consistency check. For example, the server attempted to read record file, but detected a memory parity error. The client can retry the request, but service may be required on the server device
0x0A	Gateway path unavailable	For specialized use in conjunction with gateways, hubs, switches and similar network devices, to indicate that the gateway was unable to allocate an internal communication path from the input port to the output port for processing the request. This usually means that the gateway is misconfigured or overloaded
0x0B	Gateway target device failed to respond	For specialized use in conjunction with gateways, hubs, switches and similar network devices, to indicate that no response was obtained from the target device. This usually means that the device is not present on the network

5.2.7 Client to server unconfirmed service request

The format is as in Figure 5. These services are used for broadcast. Only a small number of function codes allow broadcast.

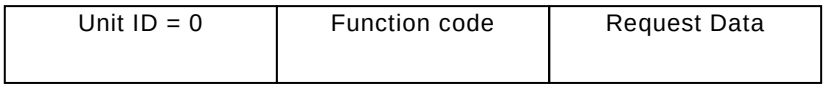

Figure 5 – Client to server unconfirmed service request

5.3 Service-specific APDU structures

5.3.1 Read Discretes FAL PDU

5.3.1.1 Request primitive

Service identifier, function code = 2 (0x02).

This function code is used to read the status of 1 to 2 000 discrete inputs in a remote device. The request PDU specifies the starting address, i.e. the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore discrete inputs numbered 1 to 16 are addressed as 0 to 15. The starting address can be from 0x0000 to 0xFFFF.

The format is given in Table 3.

Table 3 – Read discretes request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 2 (0x02).
Address of first discrete	Unsigned16	Address of the first discrete input. Allowed values: 0x0000 to 0xFFFF
Quantity of discretes	Unsigned16	Quantity of discretes. Allowed values: 1 to 2 000 (0x7D0)

5.3.1.2 Response primitive

The format is given in Table 4.

Table 4 – Read discretes response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 2 (0x02). Echo of requested
Data octets count	Unsigned16	Number of octets read
Data	Status bit sequence	Status values of the discretes read

The field data contains n octets where n is the number in the field data octets count.

The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The lsb (least significant bit) of the first data octet contains the input addressed in the query. The other inputs follow toward the high order end of this octet, and from low order to high order in subsequent octets.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data octet shall be padded with zeros (toward the high order end of the octet). The data octets count field specifies the quantity of octets of returned inputs data, n (including the padded octet, if any).

5.3.2 Read Coils FAL PDU

5.3.2.1 Request primitive

Service identifier, function code = 1 (0x01).

This function code is used to read from 1 to 2 000 coils in a remote device. The request PDU specifies the starting address, i.e. the address of the first coil specified, and the number of coils. In the PDU, coils are addressed starting at zero. Therefore coils numbered 1 to 16 are addressed as 0 to 15. The starting address can be from 0x0000 to 0xFFFF.

The format is given in Table 5.

Table 5 – Read coils request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 1 (0x01).
Address of first coil	Unsigned16	Address of the first coil. Allowed values: 0x0000 to 0xFFFF
Quantity of coils	Unsigned16	Quantity of coils. Allowed values: 1 to 2 000 (0x7D0)

5.3.2.2 Response primitive

The format is given in Table 6.

Table 6 – Read coils response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 1 (0x01). Echo of requested
Data octets count	Unsigned16	Number of octets read
Data	Status bit sequence	Status values of the discretes read

The field data contains n octets where n is the number in the field data octets count.

The coils in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The lsb (least significant bit) of the first data octet contains the input addressed in the query. The other coils follow toward the high order end of this octet, and from low order to high order in subsequent octets.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data octet shall be padded with zeros (toward the high order end of the octet). The data octets count field specifies the quantity of octets of returned inputs data, n (including the padded octet, if any).

5.3.3 Write Single Coil FAL PDU

5.3.3.1 Request primitive

Service identifier, function code = 5 (0x05).

This function code is used to write a single coil to either ON or OFF in a remote device.

The requested ON/OFF status is specified by a constant in the request data field. A value of 0xFF00 requests the output to be ON. A value of 0x0000 requests it to be OFF. All other values are illegal and do not affect the output.

The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The format is given in Table 7.

Table 7 – Write single coil request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 5 (0x05).
Address of coil	Unsigned16	Address of the coil. Allowed values: 0x0000 to 0xFFFF
Data single coil	Status flag	Single coil status value that has to be written. Allowed values: 0xFF00 or 0x0000

5.3.3.2 Response primitive

The format is given in Table 8.

Table 8 – Write single coil response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested.
Function code	Unsigned8	Service identifier, function code = 5 (0x05). Echo of requested.
Address of coil	Unsigned16	Address of the coil. Echo of requested.
Data single coil	Status flag	Single coil status value that had to be written. Echo of requested.

The normal response is an echo of the request, returned after the coil state has been written.

5.3.4 Write Multiple Coils FAL PDU

5.3.4.1 Request primitive

Service identifier, function code = 15 (0x0F).

This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote device.

The Request PDU specifies the coil references to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The requested ON/OFF coils status is specified by the contents of the request data field. A logical '1' in a bit position of the field requests the corresponding output to be ON. A logical '0' requests it to be OFF.

The coils in the request message are packed as one input per bit of the data field. If the specified quantity of coils is not a multiple of eight, the remaining bits in the final data octet of data shall be padded with zeros (toward the high order end of the octet). The octet count field specifies the quantity of octets of data, n (including the padded octet, if any). The field data contains the n data octets, where n is the number in the field octet count.

The format is given in Table 9.

Table 9 – Write multiple coils request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 15 (0x0F).
Address of first coil	Unsigned16	Address of the first coil. Allowed values: 0x0000 to 0xFFFF
Quantity of coils	Unsigned16	Quantity of coils to be written. Allowed values: 1 to 1 968 (0x7B0), must be consistent with the Data octets count and the Data parameters
Data octets count	Unsigned16	Number of octets carrying the coil status values to be written. Allowed values: 1 to 246, must be consistent with the Quantity of coils and the Data parameters
Data	Status bit sequence	This parameter shall be used to specify the coil status values that have to be written. Allowed values:Must be consistent with the Quantity of coils and the Data octets count parameters

5.3.4.2 Response primitive

The format is given in Table 10.

Table 10 – Write multiple coils response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 15 (0x0F)
Address of first coil	Unsigned16	Address of the first coil. Echo of requested
Quantity of coils	Unsigned16	Quantity of coils. Echo of requested

The normal response is an echo of the requested parameters Unit ID, function code, address of first coil, and quantity of coils.

5.3.5 Broadcast Write Single Coil FAL PDU

5.3.5.1 Request primitive

Service identifier, function code = 5 (0x05); Unit ID = 0.

This function code is used to write a single coil to either ON or OFF in all the Unit ID addressable servers, by specifying Unit ID = 0.

This is an unconfirmed service.

The requested ON/OFF status is specified by a constant in the request data field. A value of 0xFF00 requests the output to be ON. A value of 0x0000 requests it to be OFF. All other values are illegal and do not affect the output.

The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The format is given in Table 11.

Table 11 – Broadcast write single coil request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 0
Function code	Unsigned8	Service identifier, function code = 5 (0x05)
Address of coil	Unsigned16	Address of the coil. Allowed values: 0x0000 to 0xFFFF
Data single coil	Status flag	Single coil status value that has to be written. Allowed values: 0xFF00 or 0x0000

5.3.6 Broadcast Write Multiple Coils FAL PDU

5.3.6.1 Request primitive

Service identifier, function code = 15 (0x0F); Unit ID = 0.

This function code is used to force each coil in a sequence of coils to either ON or OFF in all the Unit ID addressable servers, by specifying Unit ID = 0.

This is an unconfirmed service.

The Request PDU specifies the coil references to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The requested ON/OFF coils status is specified by the contents of the request data field. A logical '1' in a bit position of the field requests the corresponding output to be ON. A logical '0' requests it to be OFF.

The coils in the request message are packed as one input per bit of the data field. If the specified quantity of coils is not a multiple of eight, the remaining bits in the final data octet of data shall be padded with zeros (toward the high order end of the octet). The octet count field specifies the quantity of octets of data, n (including the padded octet, if any). The field data contains the n data octets, where n is the number in the field octet count.

The format is given in Table 12.

Table 12 – Broadcast write multiple coils request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 0
Function code	Unsigned8	Service identifier, function code = 15 (0x0F)
Address of first coil	Unsigned16	Address of the first coil. Allowed values: 0x0000 to 0xFFFF
Quantity of coils	Unsigned16	Quantity of coils to be written. Allowed values: 1 to 1 968 (0x7B0), must be consistent with the Data octets count and the Data parameters
Data octets count	Unsigned16	Number of octets carrying the coil status values to be written. Allowed values: 1 to 246, must be consistent with the Quantity of coils and the Data parameters
Data	Status bit sequence	This parameter shall be used to specify the coil status values that have to be written. Allowed values:Must be consistent with the Quantity of coils and the Data octets count parameters

5.3.7 Read Input Registers FAL PDU

5.3.7.1 Request primitive

Service identifier, function code = 4 (0x04).

This function code is used to read from 1 to 125 input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1 to 16 are addressed as 0 to 15.

The format is given in Table 13.

Table 13 – Read input registers request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 4 (0x04)
Address of input register to read	Unsigned16	Address of input register.to read. Allowed values: 0x0000 to 0xFFFF
Quantity of input registers	Unsigned16	Quantity of input registers to read. Allowed values: 1 to 125 (0x7D)

5.3.7.2 Response primitive

The format is given in Table 14.

Table 14 – Read input registers response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 4 (0x04). Echo of requested
Data octets count	Unsigned16	Number of octets read
Data	Array of Unsigned16	Input registers values read

The response parameter data contains n octets, where n is the number in the response parameter data octet count, equal to twice the requested quantity of input registers.

The register data in the response parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register value bits and the second octet contains the low order register value bits (big-endian convention).

5.3.8 Read Holding Registers FAL PDU

5.3.8.1 Request primitive

Service identifier, function code = 3 (0x03).

This function code is used to read from 1 to 125 holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1 to 16 are addressed as 0 to 15.

The format is given in Table 15.

Table 15 – Read holding registers request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 3 (0x03)
Address of first holding register to read	Unsigned16	Address of the first holding register to read. Allowed values: 0x0000 to 0xFFFF
Quantity of holding registers to read	Unsigned16	Quantity of holding registers to read. Allowed values: 1 to 125 (0x7D)

5.3.8.2 Response primitive

The format is given in Table 16.

Table 16 – Read holding registers response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 3 (0x03). Echo of requested
Data octets count	Unsigned16	Number of octets read
Data	Array of Unsigned16	Holding registers values read

The response parameter data contains n octets, where n is the number in the response parameter data octet count, equal to twice the requested quantity of holding registers.

The register data in the response parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register value bits and the second octet contains the low order register value bits (big-endian convention).

5.3.9 Write Single Holding Register FAL PDU

5.3.9.1 Request primitive

Service identifier, function code = 6 (0x06).

This function code is used to write a single holding register in a remote device.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

The register data in the request parameter data is packed as two octets per register. For each register, the first octet contains the high order register bits and the second octet contains the low order register bits (big-endian convention).

The format is given in Table 17.

Table 17 – Write single holding register request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 6 (0x06)
Address of holding register to write	Unsigned16	Address of the holding register to write. Allowed values: 0x0000 to 0xFFFF
Data	Unsigned16	Holding register value to be written. Allowed valies: 0x0000 to 0xFFFF

5.3.9.2 Response primitive

The format is given in Table 18.

Table 18 – Write single holding register response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 6 (0x06). Echo of requested
Address holding register to write	Unsigned16	Address of the holding register to write. Echo of requested
Data	Unsigned16	Holding register value to be written. Echo of requested

The normal response is an echo of the request, returned after the register contents have been written.

5.3.10 Write Multiple Holding Registers FAL PDU

5.3.10.1 Request primitive

Service identifier, function code = 16 (0x10).

This function code is used to write from 1 to 123 holding registers in a remote device.

The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1 to 16 are addressed as 0 to 15.

The values to be written are specified in the request data parameter. The register data in the request parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register bits and the second octet contains the low order register bits (big-endian convention).

The format is given in Table 19.

Table 19 – Write multiple holding registers request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 16 (0x10)
Address of first holding register to write	Unsigned16	Address of the first holding register to write. Allowed values: 0x0000 to 0xFFFF
Quantity of holding registers to write	Unsigned16	Quantity of holding registers to write. Allowed values: 1 to 123 (0x7B), must be consistent with the Data octets count and the Data parameters
Data octets count	Unsigned16	Number of octets to write. Allowed values: 1 to 246, must be consistent with the Quantity of holding registers to write and the Data parameters
Data	Array of Unsigned16	Holding registers values to write. Allowed values: 1 to 123 elements, must be consistent with the Quantity of holding registers to write and the Data octets count parameters. Element values can be from 0x0000 to 0xFFFF

5.3.10.2 Response primitive

The format is given in Table 20.

Table 20 – Write multiple holding registers response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 16 (0x10). Echo of requested
Address of first holding register to write	Unsigned16	Address of the first holding register to write. Echo of requested
Quantity of holding registers to write	Unsigned16	Quantity of holding registers to write. Echo of requested

The normal response returns the requested parameters Unit ID, function code, address of first holding register to write, and quantity of registers to write.

5.3.11 Mask Write Holding Register FAL PDU

5.3.11.1 Request primitive

Service identifier, function code = 22 (0x16).

This function code is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register. This is done according to Equation (1).

$$new\_content = (old\_content \wedge AND\_Mask) \vee (OR\_Mask \wedge \neg AND\_Mask) \quad (1)$$

The request specifies the holding register to be written, the data to be used as the AND mask, and the data to be used as the OR mask. Registers are addressed starting at zero. Therefore registers 1 to 16 are addressed as 0 to 15.

The format is given in Table 19.

Table 21 – Mask write holding register request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 22 (0x16)
Address of holding register to change	Unsigned16	Address of the holding register to change. Allowed values: 0x0000 to 0xFFFF
AND Mask	Unsigned16	Binary mask AND_Mask that contributes to the new content of the holding register to change according to equation (1). Allowed values: 0x0000 to 0xFFFF
OR Mask	Unsigned16	Binary mask OR_Mask that contributes to the new content of the holding register to change according to equation (1). Allowed valies: 0x0000 to 0xFFFF

5.3.11.2 Response primitive

The format is given in Table 22.

Table 22 – Mask write holding register request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 22 (0x16). Echo of requested
Address of holding register to change	Unsigned16	Address of the holding register to change. Echo of requested
AND Mask	Unsigned16	Binary mask AND_Mask that contributes to the new content of the holding register to change according to equation (1). Echo of requested
OR Mask	Unsigned16	Binary mask OR_Mask that contributes to the new content of the holding register to change according to equation (1). Echo of requested

The normal response is an echo of the request, returned after the register contents have been written.

5.3.12 Read/Write Holding Registers FAL PDU

5.3.12.1 Request primitive

Service identifier, function code = 23 (0x17).

This function code performs a combination of one read operation and one write operation on holding registers in a single service.

The write operation is performed before the read operation.

Holding registers are addressed starting at zero. Therefore holding registers 1 to 16 are addressed in the PDU as 0 to 15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The octet count specifies the number of octets to follow in the write data field.

The parameter data contains n octets, where n is the number in the parameter data octets count, equal to twice the requested quantity of holding registers to write parameter.

The values to be written are specified in the request data parameter. The register data in the request parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register bits and the second octet contains the low order register bits (big-endian convention).

The format is given in Table 23.

Table 23 – Read/Write multiple holding registers request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 23 (0x17)
Address of first holding register to read	Unsigned16	Address of the first holding register to read. Allowed values: 0x0000 to 0xFFFF
Quantity of holding registers to read	Unsigned16	Quantity of holding registers to read. Allowed values: 1 to 125 (0x7D).
Address of first holding register to write	Unsigned16	Address of the first holding register to write. Allowed values: 0x0000 to 0xFFFF
Quantity of holding registers to write	Unsigned16	Quantity of holding registers to write. Allowed values: 1 to 121 (0x79), must be consistent with the Data octets count and the Data parameters
Data octets count	Unsigned16	Number of octets to write. Allowed values: 1 to 242, must be consistent with the Quantity of holding registers to write and the Data parameters
Data	Array of Unsigned16	Holding registers values to write. Allowed values: 1 to 123 elements, must be consistent with the Quantity of holding registers to write and the Data octets count parameters. Element values can be from 0x0000 to 0xFFFF

5.3.12.2 Response primitive

The format is given in Table 16.

Table 24 – Read/Write multiple holding registers response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 23 (0x17). Echo of requested
Data octets count	Unsigned16	Number of octets read
Data	Array of Unsigned16	Holding registers values read

The response parameter data contains n octets, where n is the number in the response parameter data octet count, equal to twice the requested quantity of holding registers.

The register data in the response parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register value bits and the second octet contains the low order register value bits (big-endian convention).

5.3.13 Read FIFO FAL PDU

5.3.13.1 Request primitive

Service identifier, function code = 24 (0x18).

This function code is used to read a bounded number of holding registers, organized to facilitate a FIFO policy. The bounded number is a-priori unknown, and it is part of the response. The bound is 32 registers: the register containing the above number plus up to 31 following registers.

The format is given in Table 25.

Table 25 – Read FIFO request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 24 (0x18)
Address of FIFO queue	Unsigned16	Address of the holding register that contains the count of the FIFO queue data holding registers to follow. Allowed values: 0x0000 to 0xFFFF

5.3.13.2 Response primitive

The format is given in Table 26.

Table 26 – Read FIFO response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested
Function code	Unsigned8	Service identifier, function code = 3 (0x03). Echo of requested
Data octets count	Unsigned16	Number of octets read, including the octets of the FIFO queue count
FIFO queue count	Unsigned16	Number of data holding registers of the FIFO queue, read in the Data parameter. The FIFO queue count holding register itself is not included
Data	Array of Unsigned16	Holding registers values read

In a normal response, the data octet count shows the quantity of octets to follow, including the FIFO queue count octets and the data octets.

The FIFO queue count is the quantity of data registers in the queue (not including the FIFO queue count register itself).

The register data in the response parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register value bits and the second octet contains the low order register value bits (big-endian convention).

5.3.14 Broadcast Write Single Holding Register FAL PDU

5.3.14.1 Request primitive

Service identifier, function code = 6 (0x06); Unit ID = 0.

This function code is used to write a single holding register in all the Unit ID addressable servers, by specifying Unit ID = 0.

This is an unconfirmed service.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

The register data in the request parameter data is packed as two octets per register. For each register, the first octet contains the high order register bits and the second octet contains the low order register bits (big-endian convention).

The format is given in Table 27.

Table 27 – Broadcast write single holding register request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 0
Function code	Unsigned8	Service identifier, function code = 6 (0x06)
Address of holding register to write	Unsigned16	Address of the holding register to write. Allowed values: 0x0000 to 0xFFFF
Data	Unsigned16	Holding register value to be written. Allowed valies: 0x0000 to 0xFFFF

5.3.15 Broadcast Write Multiple Holding Registers FAL PDU

5.3.15.1 Request primitive

Service identifier, function code = 16 (0x10); Unit ID = 0.

This function code is used to write from 1 to 123 holding registers in all the Unit ID addressable servers, by specifying Unit ID = 0.

This is an unconfirmed service.

The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1 to 16 are addressed as 0 to 15.

The values to be written are specified in the request data parameter. The register data in the request parameter data elements is packed as two octets per register value. For each register, the first octet contains the high order register bits and the second octet contains the low order register bits (big-endian convention).

The format is given in Table 28.

Table 28 – Broadcast write multiple holding registers request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 0
Function code	Unsigned8	Service identifier, function code = 16 (0x10)
Address of first holding register to write	Unsigned16	Address of the first holding register to write. Allowed values: 0x0000 to 0xFFFF
Quantity of holding registers to write	Unsigned16	Quantity of holding registers to write. Allowed values: 1 to 123 (0x7B), must be consistent with the Data octets count and the Data parameters
Data octets count	Unsigned16	Number of octets to write. Allowed values: 1 to 246, must be consistent with the Quantity of holding registers to write and the Data parameters
Data	Array of Unsigned16	Holding registers values to write. Allowed values: 1 to 123 elements, must be consistent with the Quantity of holding registers to write and the Data octets count parameters. Element values can be from 0x0000 to 0xFFFF.

5.3.16 Read File Record FAL PDU

5.3.16.1 Request primitive

Service identifier, function code = 20 (0x14).

This function code is used to read multiple records from one or more files.

The format is given in Table 29.

Table 29 – Read file record request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 20 (0x14).
Sub-requests all-elements octets count	Unsigned8	Total count of octets (excluding itself) of all the following sub-requests. Allowed values: 7 (0x07) to 245 (0xF5). The minimum is obtained when there is only one Sub-request element, and the maximum is obtained when there are 35 Sub-request elements. The upper bound is dictated by the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets). A correct Read File Record request will also have to ensure that the total size of the requested response, including all requested records, does not exceed this upper bound.
Sub-request 1 reference type	Unsigned8	Part of a sub-request element used to specify the reference type. Allowed values: In the context of this service the only allowed value is 6.
Sub-request 1 file number	Unsigned16	Part of a sub-request element used to specify the file number. Allowed values: The lowest file number is 1. The highest file number should be 10 (0x0A). NOTE 1 While it is allowed for the file fumber to be in the range 1 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if the file number is greater than 10 (0x0A).
Sub-request 1 record number	Unsigned16	Part of a Sub-request element used to specify the record number, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the address of the first register of the record. Allowed values: Each file but the last should contain 10 000 registers, with the last file allowed to have less. This provides for records addressed from 0x0000 to 0x270F (0 to 9 999 decimal), at most. As a consequence, the Record Number should be in the range 0x0000 to 0x270F. NOTE 2 While it is allowed for any file to have more or less than 10 000 registers, with a maximum of 65 536 (0x10000), and consequently to have records addressed from 0x0000 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if each file but the last does not have 10 000 registers, with the last file allowed to have less. NOTE 3 Differently from other APOs, like discretes, coils, input registers and holding registers, where the lowest addressable instance is known to an application as 1-based and it is addressed in the protocol as 0-based, the lowest addressable record is record 0, known to an application as 0-based, and it is addressed in the protocol using a 0-based register address.
Sub-request 1 record length	Unsigned16	Part of a sub-request element used to specify the record length, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the length of the record in number of registers. Allowed values: For a given record number, the record length, in number of registers, must result in a record contained in the file. Moreover, such record length, in combination with all the other parts of the request, shall not produce a response that exceeds the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets).
Sub-request 2		
Sub-request n		

5.3.16.2 Response primitive

The format is given in Table 30.

Table 30 – Read file record response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested.
Function code	Unsigned8	Service identifier, function code = 20 (0x14). Echo of requested.
Sub-requests all-elements octets count	Unsigned8	Total count of octets (excluding itself) of all the following sub-responses. Expected values: The successful value depends on the request. For any successful request the value will be in the range 4 (0x04) to 250 (0xFA). The minimum is obtained when there is only one sub-response element and that is the smaller sub-response, with a record of length 1 register. The maximum can be reached in several ways, with different combinations of number of sub-responses and record lengths, for example with 34 sub-responses each with a record of length 1 register (136 octets so far), and one additional sub-response with a record of length of 56 registers (2 + (56 * 2) = 114 octets). The upper bound is dictated by the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets). A correct read file record request will have ensured that the total size of the requested response, including all requested records, does not exceed this upper bound.
Sub-response 1 octets count	Unsigned8	Part of a sub-response element used to specify the total count of octets (excluding itself) for the sub-response. The count includes the reference type octet and the octets contained in the record data array, all together 1 + twice the record length specified in the corresponding sub-request, which is expressed in number of registers. Expected values: The successful value depends on the request. For any successful request the value will be a minimum of 3 (0x03) and a maximum of 249 (0xF9). The minimum is obtained when the requested record has the length of 1 register. The maximum is reached when the requested record has the length of 124 registers, and in this case this is the only sub-response.
Sub-response 1 reference type	Unsigned8	Part of a sub-response element used to specify the reference type. Echo of the requested.
Sub-response 1 record data array	Unsigned16	1-st register of the record data array of a sub-response element.
	Unsigned16	2-nd register of the record data array of a sub-response element.
	Unsigned16	n-th register of the record data array of a sub-response element.
Sub-response 2		
Sub-response n		

5.3.17 Write File Record FAL PDU

5.3.17.1 Request primitive

Service identifier, function code = 21 (0x15).

This function code is used to write multiple records into one or more files.

The format is given in Table 31.

Table 31 – Write file record request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 21 (0x15).
Sub-requests all-elements octets count	Unsigned8	Total count of octets (excluding itself) of all the following sub-requests. Allowed values: 9 (0x09) to 251 (0xFB). The minimum is obtained when there is only one Sub-request element, and that is the smaller sub-request, with a record of length 1 register. The maximum can be reached in several ways, with different combinations of number of sub-requests and record lengths, for example with 3 sub-requests, one with a record of length 1 register (9 octets so far), one with a record of length 113 registers (130 octets so far) and finally another one with a record length of 113 registers (for a total of 251 octets). The upper bound is dictated by the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets). A correct write file record request will have a normal response that does not exceed the upper bound, since in this case, as described below, a successful write file record response is an exact copy of the write file record request.
Sub-request 1 reference type	Unsigned8	Part of a sub-request element used to specify the reference type. Allowed values: In the context of this service the only allowed value is 6.
Sub-request 1 file number	Unsigned16	Part of a sub-request element used to specify the file number. Allowed values: The lowest file number is 1. The highest file number should be 10 (0x0A). NOTE 1 While it is allowed for the file fumber to be in the range 1 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if the file number is greater than 10 (0x0A).
Sub-request 1 record number	Unsigned16	Part of a Sub-request element used to specify the record number, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the address of the first register of the record. Allowed values: Each file but the last should contain 10 000 registers, with the last file allowed to have less. This provides for records addressed from 0x0000 to 0x270F (0 to 9 999 decimal), at most. As a consequence, the Record Number should be in the range 0x0000 to 0x270F. NOTE 2 While it is allowed for any file to have more or less than 10 000 registers, with a maximum of 65 536 (0x10000), and consequently to have records addressed from 0x0000 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if each file but the last does not have 10 000 registers, with the last file allowed to have less. NOTE 3 Differently from other APOs, like discretes, coils, input registers and holding registers, where the lowest addressable instance is known to an application as 1-based and it is addressed in the protocol as 0-based, the lowest addressable record is record 0, known to an application as 0-based, and it is addressed in the protocol using a 0-based register address.

Parameter name / field	Type	Description
Sub-request 1 record length	Unsigned16	Part of a sub-request element used to specify the record length, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the length of the record in number of registers. Allowed values: For a given record number, the record length, in number of registers, must result in a record contained in the file. Moreover, such record length, in combination with all the other parts of the request, shall not produce a response that exceeds the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets).
Sub-request 1 record data array	Unsigned16	1-st register of the record data array of a sub-request element.
	Unsigned16	2-nd register of the record data array of a sub-request element.
	Unsigned16	3-rd register of the record data array of a sub-request element.
Sub-request 2		
Sub-request n		

5.3.17.2 Response primitive

The normal response is an echo of the request. The format is given in Table 32.

Table 32 – Write file record response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested.
Function code	Unsigned8	Service identifier, function code = 21 (0x15). Echo of requested.
Sub-requests all-elements octets count	Unsigned8	Total count of octets (excluding itself) of all the following sub-requests. Echo of requested.
Sub-request 1 reference type	Unsigned8	Part of a sub-request element used to specify the reference type. Echo of requested.
Sub-request 1 file number	Unsigned16	Part of a sub-request element used to specify the file number. Echo of requested.
Sub-request 1 record number	Unsigned16	Part of a Sub-request element used to specify the record number, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the address of the first register of the record. Echo of requested.
Sub-request 1 record length	Unsigned16	Part of a sub-request element used to specify the record length, that contributes to the qualification of the record. Records are identified using the address of their first register and their length, the latter is specified in number of registers; this parameter represents the length of the record in number of registers. Echo of requested.
Sub-request 1 record data array	Unsigned16	1-st register of the record data array of a sub-request element. Echo of requested.
	Unsigned16	2-nd register of the record data array of a sub-request element. Echo of requested.
	Unsigned16	3-rd register of the record data array of a sub-request element. Echo of requested.
Sub-request 2		Echo of requested.
Sub-request n		Echo of requested.

5.3.18 Read Device Identification FAL PDU

5.3.18.1 Request primitive

Service identifier, function code/MEI Type = 43 (0x2B)/14 (0x0E).

This function code/MEI Type is used to retrieve the device identification objects.

The format is given in Table 33.

Table 33 – Read device identification request

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Allowed values: 1 to 247
Function code	Unsigned8	Service identifier, function code = 43 (0x2B).
MEI	Unsigned8	Encapsulated Interface type. Allowed values:14 (0x0E).
Read device ID code	Unsigned8	Requested device access type, that qualifies the requested information based on device categories described in Table 34 and, when supported, individually addressed object retrieval. This is illustrated in Table 35. Allowed values: As illustrated in Table 35.
Requested object ID	Unsigned8	First requested object, or the single requested object, according to the Read Device ID code. A response cannot exceed the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets). An individual object size is guaranteed to fit in the maximum size by definition. If the set of returned objects requires more octets than the maximum size, then several transactions (request/response) are needed. This is an application client responsibility. When many objects are requested, the ID of the first requested object of the first transaction must be set to 0x00. If the initial object ID is not 0x00, then in general the returned Device Identification will be incomplete. Subsequent requests within the same set of objects must set the Requested-object ID to the Next-object ID returned by the server in the previous response. If a different Requested-object ID is provided, then in general the returned Device Identification will be incomplete. When many objects are requested, if the Requested-object ID does not match any known object, the server will respond as if the object with ID 0x00 was requested, effectively restarting from the beginning if this happens in the middle of the retrieval. When the server supports the retrieval of a single object, and this is performed with a Requested-object ID that does not match any known object, the server will return an error response. Allowed values: 0x00 to 0xFF.

Table 34 – Device identification categories

Object IDs	Object name / description	Type	Presence	Category
0x00 0x01 0x02	Vendor Name Product Code Major Minor Revision	ASCII string ASCII string ASCII string	Mandatory Mandatory Mandatory	Basic
0x03 0x04 0x05 0x06 0x07 … 0x7F	Vendor URL Product Name Model Name User Application Name *Reserved*	ASCII string ASCII string ASCII string ASCII string	Optional	Regular
0x80 … 0xFF	*Private application-defined objects. The object ID range [0x80 – 0xFF] can be used by an application to define its own objects.*	Device dependent	Optional	Extended

Table 35 – Read device ID code

Value	Read device ID code
0x01	Request to retrieve the objects in the Basic Device Identification category. A stream of objects is expected
0x02	Request to retrieve the objects in the Regular Device Identification category, if any, and this implies a request for the Basic Device Identification category as well. A stream of objects is expected, with the Basic Device Identification category returned first, and the Regular Device Identification category afterward, if any
0x03	Request to retrieve the objects in the Extended Device Identification category, if any, and this implies a request for the Regular Device Identification category, if any, and for the Basic Device Identification category as well. A stream of objects is expected, with the Basic Device Identification category returned first, the Regular Device Identification category after that, if any, and finally the Extended Device Identification category, if any
0x04	Request to retrieve a specific object. If this feature is supported, a single object is expected, otherwise an error response is returned

NOTE While the returned categories are ordered as from Table 35, no assumption should be made about the order of returned objects within any category, even across service invocations. This is to avoid any extra processing load on servers that may reside in very simple devices, and to permit the best object packing when the retrieval of multiple objects requires multiple request/response transactions.

5.3.18.2 Response primitive

The format is given in Table 36.

Table 36 – Read device identification response

Parameter name / field	Type	Description
Unit ID	Unsigned8	Address of the server. Echo of requested.
Function code	Unsigned8	Service identifier, function code = 43 (0x2B). Echo of requested.
MEI	Unsigned8	Encapsulated Interface type. Echo of requested.
Read device ID code	Unsigned8	Requested device access type, that qualifies the requested information based on device categories described in Table 34 and, when supported, individually addressed object retrieval. This is illustrated in Table 35. Echo of requested.
Conformity level	Unsigned8	Actual object categories and object retrieval access type made available by the server. Its value is provided by the server in all the responses, irrespective of the requested Read Device ID code. Values are illustrated in Table 37. In the Read Device ID parameter description it was explained what is returned when dealing with objects unknown to the server. For known objects, if a Read Device ID code requests a category or a type of access that is not available on the server, then the returned objects are as from Table 38.
More-available flag	Unsigned8	Information about having or not more objects to retrieve after a request/response. It is meaningful when the Read Device ID code is one of 0x01, 0x02, or 0x03, and the response exceed the maximum size of the APDU for client/server (Unit ID + Function Code + Data = 254 octets). A value of 0x00 means that there are no more objects available, while a value of 0xFF means that more requests have to be issued to retrieve the remaining objects. The meaning of this parameter is related to the one of the Next-object ID parameter.
Next-object ID	Unsigned8	The ID of the object that has to be requested in a subsequent request when the More-available flag is 0xFF. This is a client's responsibility, and if the Next-object ID requested in the subsequent request does not match any known object, the server will respond as if the object with ID 0x00 was requested, effectively restarting from the beginning.
Number of objects	Unsigned8	Part of the Read Device Identification array parameter. Each element carries one object. Each element uniquely identifies and represents an object within the device identification address space using the Returned-object ID, the Object length and the Object value. All is described below.
Object 1 returned-object ID	Unsigned8	Part of the Objects array element, and it uniquely identifies the object. Its description is the same as for the Requested-object ID, and it is part of the stream that initiated with the Requested-object ID.
Object 1 Object length	Unsigned8	Part of the Objects array element. It describes the length of the object value, in octets.
Object 1 Object value	As from Table 34	Part of the Objects array element and contributes to the device identification.
Object 2		
…		
Object n		

Table 37 – Conformity level

Value	Read device ID code
0x01	The device supports only the Basic Device Identification category, and only stream access
0x02	In addition to the Basic Device Identification category, the device supports also the Regular Device Identification category, and only stream access
0x03	In addition to the Basic Device Identification category and to the Regular Device Identification category, the device supports also the Extended Device Identification category, and only stream access
0x81	The device supports only the Basic Device Identification category, and both stream access and individual access
0x82	In addition to the Basic Device Identification category, the device supports also the Regular Device Identification category, and both stream access and individual access
0x83	In addition to the Basic Device Identification category and to the Regular Device Identification category, the device supports also the Extended Device Identification category, and both stream access and individual access

Table 38 – Requested vs. returned known objects

Read device ID	Conformity level	Returned objects
0x01or 0x02 or 0x03	0x01	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x01	0x02	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x02 or 0x03	0x02	In addition to objects in the Basic Device Identification category, the server returns afterward also the objects in the Regular Device Identification category. Objects are returned as a stream
0x01	0x03	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x02	0x03	In addition to objects in the Basic Device Identification category, the server returns afterward also the objects in the Regular Device Identification category. Objects are returned as a stream
0x03	0x03	In addition to objects in the Basic Device Identification category and after that objects in the Regular Device Identification category, the server returns afterward also the objects in the Extended Device Identification category. Objects are returned as a stream
0x04	0x01 or 0x02 or 0x03	The server returns an illegal function error response
0x01or 0x02 or 0x03	0x81	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x01	0x82	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x02 or 0x03	0x82	In addition to objects in the Basic Device Identification category, the server returns afterward also the objects in the Regular Device Identification category. Objects are returned as a stream
0x01	0x83	The server returns the objects in the Basic Device Identification category. Objects are returned as a stream
0x02	0x83	In addition to objects in the Basic Device Identification category, the server returns afterward also the objects in the Regular Device Identification category. Objects are returned as a stream
0x03	0x83	In addition to objects in the Basic Device Identification category and after that objects in the Regular Device Identification category, the server returns afterward also the objects in the Extended Device Identification category. Objects are returned as a stream
0x04	0x81 or 0x82 or 0x83	The server returns the individually requested object

5.4 Data representation ‘on the wire’

Client/server uses a big-endian representation for addresses and data items. This means that when a numerical quantity larger than a single octet is transmitted, the most significant octet is sent first.

EXAMPLE A register is of type Unsigned16, a multiple octet quantity. If its value is 0x1234, then the first octet sent is 0x12, and the second octet sent is 0x34.

6 Abstract syntax for publish/subscribe

The abstract syntax of APDUs is combined with their transfer syntax and is specified in Clause 7.

7 Transfer syntax for publish/subscribe

7.1 General

The sending Application Layer prepares an APDU to transfer to the receiving Application Layer. It uses the parameters from the service primitives to do so. There is only one APDU format, it is the packed format produced with the help of the messenger services, which thread together APDUs from the other services. This permits flexibility and expansion, with differences encoded as content in the APDU itself.

In the context of publish/subscribe the packed APDU is also called message, and the constituent sub-APDUs are also called sub-messages. The APDU composition is illustrated using UML notation in Figure 6.

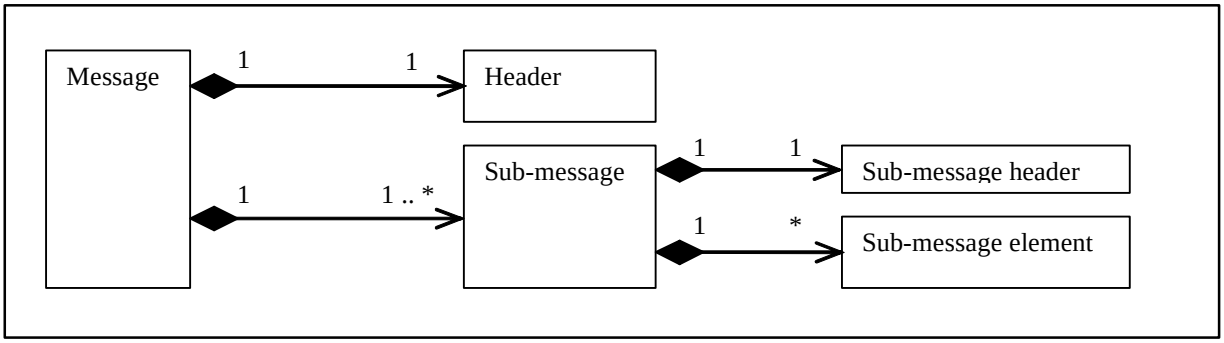

Figure 6 – Publish/subscribe APDU

7.2 APDU structure

The generic APDU is made of a leading header followed by a variable number of sub-messages. Each sub-message starts aligned on a 32-bit boundary with respect to the start of the message. The APDU details are illustrated in Table 39.

The APDU has a well-known length. This length is not sent explicitly by the publish/subscribe protocol but is part of the underlying transport with which APDUs are sent. In the case of UDP/IP, the length of the APDU is the length of the UDP payload.

Table 39 – APDU structure

fields	content, aligned on a 32 bits boundary
header fixed length 16 octets	4 octets
	4 octets
	4 octets
	4 octets
sub-message 1	starts on a 32 bits boundary and has variable length
…	…
sub-message n	starts on a 32 bits boundary and has variable length

It is convenient to see the message header as a sub-APDU itself, despite it being a header. In IEC 61158-5-15 it has been abstracted as one of the services of the messenger ASE because it has many similarities with other services there, which modify defaults set by the header. It is also different, its sub-APDU has the fixed length of 16 octets, it is a singleton and it is identified by its location instead of having a service identifier. The message header will be described as part of the service specific APDU structures, after the discussion of the sub-message structure.

The header is one of the set of logistic sub-messages: header, INFO_DST, INFO_REPLY, INFO_SRC, INFO_TS, and PAD. The publish/subscribe is a wire protocol, and these messages are all responsibility of the user application.

NOTE OMG DDS as in “Data Distribution Service for Real-Time Systems Specification, Version 1.1, December 2005” defines services that are here presented as responsibility of the publish/subscribe AL user instead. publish/subscribe is a wire protocol, and as such some functionality is deferred to the user application. This can be appreciated by the flexibility offered for implementations where foot-print is at a premium, and contributes to the pervasiveness of the approach.

Types are defined in IEC 61158-5-15. The octets for every sub-message are in the order specified up to the *flags* octet; after that, multi-octet types are placed ‘on the wire’ according to the endian-ness flag, that may modify the ordering on a per sub-message basis.

7.3 Sub-message structure

7.3.1 General

The general structure of each sub-message in a message is as illustrated in Table 40.

Table 40 – Sub-message structure

Parameter name / Field	Type	Description
Sub-message ID	Unsigned8	This is the service identifier encoding.
Flags	OctetString, 1 octet	Flags encode some of the parameters.
OctetsToNextHeader	Unsigned16	Parameter that allows the variable length sub-messages composition.
Sub-message specific content	OctetString, variable length	Described with the service specific APDU structures

7.3.2 Service identifiers

The sub-message ID octet carries the service identifier encoding. IDs 0x00 to 0x7F (inclusive) are protocol-specific. They are defined as part of the publish/subscribe protocol. The major version 1 of publish/subscribe defines the sub-message IDs as reported in Table 41.

Table 41 – Publish/subscribe service identifier encoding

Encoding	Service
0x01	PAD
0x02	VAR
0x03	Issue
0x04 – 0x05	*Reserved*
0x06	ACK
0x07	Heartbeat
0x08	GAP
0x09	INFO_TS
0x0A – 0x0B	*Reserved*
0x0C	INFO_SRC
0x0D	INFO_REPLY
0x0E	INFO_DST
0x0F – 0x7F	*Reserved*
0x80 – 0XFF	*available to be used for vendor specific extensions; their interpretation is dependent on the vendorID encoded in the message header*

The meaning of the sub-message IDs cannot be modified in this major version (1). Additional sub-messages can be added in higher minor versions. Sub-messages with ID's 0x80 to 0xFF (inclusive) are vendor-specific; they will not be defined by the protocol. Their interpretation is dependent on the *vendorId* that is current when the sub-message is encountered. The description of the header will specify how the current *vendorId* is determined.

7.3.3 Flags

The least-significant bit (lsb) of the flags is always present in all sub-messages and represents the endian-ness used to encode the information in the sub-message. E=0 means big-endian, E=1 means little-endian. This is on a per sub-message basis.

This is the only flag that has a dedicated position, the lsb position, in the flags octet.

Other bits in the flags octet have interpretations that depend on the type of sub-message.

In the following descriptions of the sub-messages, the character 'X' is used to indicate a flag that is unused in version 1.0 of the protocol. publish/subscribe implementations of version 1.0 should set these to zero when sending and ignore these when receiving. Higher minor versions of the protocol can use these flags.

7.3.4 OctetsToNextHeader

The final two octets of the sub-message header contain the number of octets from the first octet of the contents of the sub-message until the first octet of the header of the next sub-message. The representation of this field is a Common Data Representation (CDR) unsigned short (ushort). CDR is documented in “Common Object Request Broker Architecture: Core Specification”, and in this specification in 7.6 for the part regarding publish/subscribe.

If the sub-message is the last one in the message, the *octetsToNextHeader* field contains either the number of octets remaining in the message or the sentinel 0. The meaning of the sentinel is that the length of the content of the last message has to be derived by other means (computed from the lengths of all previous sub-messages and the length of the all message). This allows for a last sub-message larger than what can be specified in the encoding Unsigned16. In general, due to alignment requirements, the *octetsToNextHeader* field may be larger than the length of the current sub-message data.

7.4 APDU interpretation

7.4.1 General

The interpretation and meaning of a sub-message/sub-APDU within a message/APDU may depend on the previous sub-messages within that same message. Therefore the receiver of a message must maintain state from previously deserialized sub-messages in the same message.

Table 42 lists attributes that will be encountered examining the service specific APDU structures; these attributes may be modally changed by the parameters of some service specific APDUs and affect the interpretation of the following ones. Types are defined in IEC 61158-5-15.

Table 42 – Attributes changed modally and affecting APDUs interpretations

Attribute name	Type	Description
sourceVersion	ProtocolVersion	The major and minor version with which the following sub-messages need to be interpreted
sourceVendorID	VendorID	The vendor identification with which the following vendor-specific extensions need to be interpreted
sourceHostID, sourceAppID	HostID, AppID	The originator's host and application identifiers. The following sub-messages need to be identified as if they are coming from this host and application
destHostID, destAppID	HostID, AppID	The destination's host and application identifiers. The following sub-messages need to be identified as if they are meant for this host and application
unicastReplyIPAddress, unicastReplyPort	IPAddress, Port	An explicit IP address and port that provides an additional direct way for the receiver to reply directly to the originator over unicast
multicastReplyIPAddress, multicastReplyPort	IPAddress, Port	An explicit IP address and port that provides an additional direct way for the receiver to reach the originator (and potentially many others) over multicast
haveTimestamp, timestamp	Boolean, NtpTime	The timestamp applying to all the following sub-messages

7.4.2 Rules

The following algorithm outlines the rules that a receiver of any message must follow:

a) If a 4-octets sub-message header cannot be read, the rest of the message is considered invalid;

b) The last two octets of a sub-message header, the *octetsToNextHeader* field, contains the number of octets to the next sub-message. If this field is invalid, the rest of the message is invalid;

c) The first octet of a sub-message header is the *sub-messageID*. A sub-message with an unknown ID must be ignored and parsing must continue with the next sub-message. Concretely: an implementation of publish/subscribe 1.0 must ignore any sub-messages with IDs that are outside of the sub-messageID list used by version 1.0. IDs in the vendor-specific range coming from a *vendorID* that is unknown must be ignored and parsing must continue with the next sub-message;

d) The second octet of a sub-message header contains flags; unknown flags should be skipped. An implementation of publish/subscribe 1.0 should skip all flags that are marked as 'X' (unused) in the protocol;

e) A valid *octetsToNextHeader* field must *always* be used to find the next sub-message, even for sub-messages with unknown IDs;

f) A known but invalid sub-message invalidates the rest of the message. 7.5.1 through 7.5.10.1 each describe a known sub-message and when it should be considered invalid.

The reception of a valid message header and/or sub-message has two effects:

— It can change the state of the receiver; this state influences how the following sub-messages in the message are interpreted. 7.5.1 through 7.5.10.1 show how the state changes for each sub-message. In this version of the protocol, only the Header and

the sub-messages INFO_SRC, INFO_REPLY and INFO_TS change the state of the receiver;

— The sub-message, interpreted within the message, has a logical interpretation: it encodes one of the five basic publish/subscribe services: ACK, GAP, HEARTBEAT, ISSUE or VAR, beside the logistic services.

7.5 Service specific APDU structures

7.5.1 Issue FAL PDU

7.5.1.1 Request primitive

Service identifier, sub-message ID = 3 (0x03).

This sub-message is used by a publisher to publish user data for one or more subscribers.

This is an unconfirmed service.

The format is given in Table 43, Figure 7 and Table 44.

Table 43 – Issue request

Parameter name / Field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 3 (0x03)
Flags	Octet string, 1 octet, see Figure 7 and Table 44	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
readerObjectID	ObjectID	Used to specify the subscription GUID <destHostID, destAppID, Issue.readerObjectID> for which the Issue service is meant. The Issue.readerObjectID can be OBJECTID_UNKNOWN, in which case the Issue service applies to all Subscriptions within the application <destHostId, destAppId>
writerObjectID	ObjectID	Used to specify the Publication GUID <sourceHostID, sourceAppID, Issue.writerObjectID> that originated the Issue
issueSeqNumber	SequenceNumber	Used to specify the sequence number identifying the Issue
parameterSequence	ParameterSequence	Conditional to the value of the hasParameterSequence flag; it shall be used to provide a variable list of operational Issue parameters, and allows for publish/subscribe extensions
issueData	Type is communicated by configuration or by convention on the topic specification or by the discovery mechanism	Used to specify the actual AI user data in this Issue

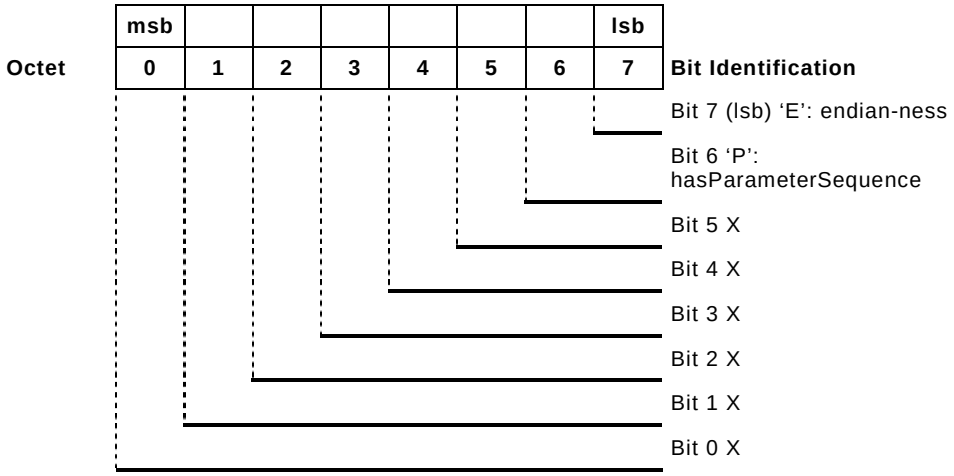

Figure 7 – Flags of issue request

Table 44 – Meaning of issue request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
hasParameterSequence	Specifies if there is or not a parameterSequence parameter. Values: {NO, 0}, {YES, 1}

7.5.1.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information;

— *issueSeqNumber* is either not strictly positive (1,2,...) or is not SEQUENCE_NUMBER_UNKNOWN;

— the parameter sequence is invalid.

7.5.1.3 Change in state of the receiver

None.

7.5.1.4 Logical interpretation

Table 45 – Interpretation of issue

Attribute	Value	Description
subscriptionGUID	<destHostid, destAppId, ISSUE.readerObjectId>	The Subscription for which the ISSUE is meant. The ISSUE.readerObjectId can be OBJECTID_UNKNOWN, in which case the ISSUE applies to all Subscriptions within the Application <destHostId, destAppId>
publicationGUID	<sourceHostId, sourceAppId, ISSUE.writerObjectId>	The Publication object that originated this issue
issueSeqNumber	ISSUE.issueSeqNumber	The sequence number of this issue; this should either be a strictly positive number (1,2,3,...) or the special sequence-number SEQUENCENUMBER_UNKNOWN. The latter may be used by a simple publication that does not number consecutive issues
(parameters)	ISSUE.parameters (iff ISSUE.P==1)	(optional) This is present iff P == 1. These parameters will allow future extensions of the protocol
ACKIPAddressPortList	{ unicastReplyIPAddress:unicastReplyPort }	The destinations to which the Publication can send an ACK message in response to this ISSUE
(timestamp)	timestamp (present iff haveTimestamp == true)	(optional) Timestamp of this issue
issueData	ISSUE.issueData	The actual user data in this issue

7.5.2 Heartbeat FAL PDU

7.5.2.1 Request primitive

Service identifier, sub-message ID = 7 (0x07).

This sub-message is used to probe a reader's presence, and to inform one or more readers about a writer's available information via sequence numbers.

This is an unconfirmed service.

The format is given in Table 46, Figure 8 and Table 47.

Table 46 – Heartbeat request

Parameter name / Field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 7 (0x07)
Flags	Octet string, 1 octet, see Figure 8 and Table 47	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
readerObjectID	ObjectID	Used to specify the reader GUID <destHostID, destAppID, Heartbeat.readerObjectID> for which the Heartbeat service is meant. The Heartbeat.readerObjectID can be OBJECTID_UNKNOWN, in which case the Heartbeat service applies to all readers within the application <destHostId, destAppId>
writerObjectID	ObjectID	Used to specify the writer GUID <sourceHostID, sourceAppID, Heartbeat.writerObjectID> that originated the Heartbeat
firstSeqNumber	SequenceNumber	Used to specify the first sequence number that is still available and meaningful in the writer with writerObjectID. This field must be greater than or equal to zero. If it is equal to SEQUENCE_NUMBER_NONE, the writer has no data available
lastSeqNumber	SequenceNumber	Used to specify the last sequence number that is available and in the writer with writerObjectID. This field must be greater than or equal to firstSeqNumber. If firstSeqNumber is equal to SEQUENCE_NUMBER_NONE, then lastSeqNumber must also be SEQUENCE_NUMBER_NONE

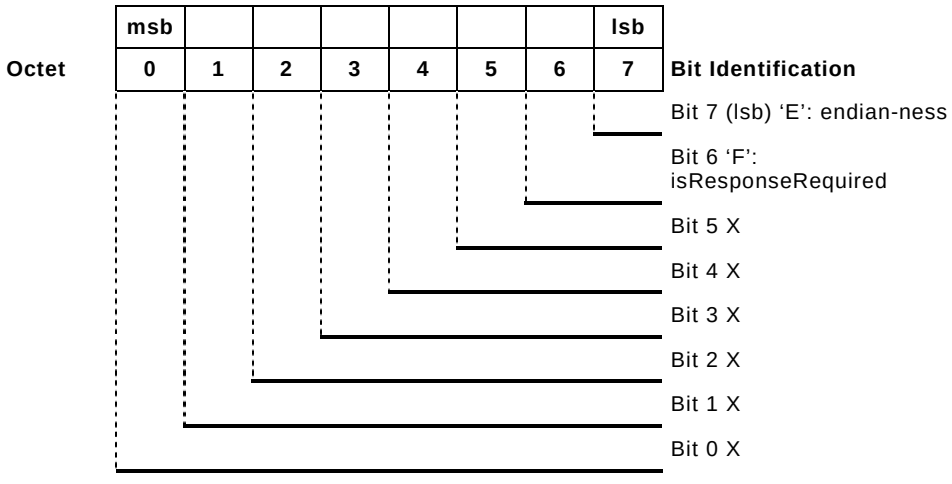

Figure 8 – Flags of heartbeat request

Table 47 – Meaning of heartbeat request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
responseIsNotRequired	Specifies if the application sending the Heartbeat requires or not a response. Known with 'F', that stands for 'Final'. Values: {FALSE, it IS required, 0}, {TRUE, it IS NOT required, 1}

7.5.2.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information;

— *firstSeqNumber* is less than 0;

— *lastSeqNumber* is less than 0;

— *lastSeqNumber* is strictly less than *firstSeqNumber*.

7.5.2.3 Change in state of the receiver

None.

7.5.2.4 Logical interpretation

Table 48 – Interpretation of heartbeat

Attribute	Value	Description
FINAL-bit	HEARTBEAT.F	When the F-bit is set, the application sending the HEARTBEAT does not require a response
readerGUID	<destHostid, destAppId, HEARTBEAT.readerObjectId>	The Reader to which the heartbeat applies. The HEARTBEAT.readerObjectId can be OBJECTID_UNKNOWN, in which case the HEARTBEAT applies to all Readers of that *writerGUID* within the Application <destHostId, destAppId>
writerGUID	<sourceHostId, sourceAppId, HEARTBEAT.writerObjectId>	The Writer to which the HEARTBEAT applies
ACKIPAddressPortList	{ unicastReplyIPAddress:unicastReplyPort }	An additional list of destinations where responses (ACKs) to this sub-message can be sent
firstSeqNumber	HEARTBEAT.firstSeqNumber	The first sequence number, *firstSeqNumber*, that is still available and meaningful in the *writerObject*. This field must be greater than or equal to zero. If it is equal to SEQUENCE_NUMBER_NONE, the Writer has no data available
lastSeqNumber	HEARTBEAT.lastSeqNumber	The last sequence number, *lastSeqNumber*, that is available in the Writer. This field must be greater than or equal to *firstSeqNumber*. If *firstSeqNumber* is SEQUENCE_NUMBER_NONE, *lastSeqNumber* must also be SEQUENCE_NUMBER_NONE

7.5.3 VAR FAL PDU

7.5.3.1 Request primitive

Service identifier, sub-message ID = 2 (0x02).

This sub-message is used by CSTWriters to publish metadata for CSTReaders, in this case information about the attributes of a network object. This information is part of a composite state.

This is an unconfirmed service.

The format is given in Table 49, Figure 9 and Table 50.

Table 49 – VAR request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 2 (0x02)
Flags	Octet string, 1 octet, see Figure 9 and Table 50	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4 Allowed values: 0x0000 to 0xFFFF
readerObjectID	ObjectID	Used to specify the reader GUID <destHostID, destAppID, VAR.readerObjectID> for which the VAR service is meant. The VAR.readerObjectID can be OBJECTID_UNKNOWN, in which case the VAR service applies to all readers of the writerObjectID within the application <destHostId, destAppId>
writerObjectID	ObjectID	Used to specify the CSTWriter GUID <sourceHostID, sourceAppID, VAR.writerObjectID> that originated the VAR
hostID	HostID	Conditional to the value of the hasHostIDandAppID flag; when present, combined with the appID and objectID parameters, shall be used to specify the GUID of the object the information carried by this VAR is about
appID	AppID	Conditional to the value of the hasHostIDandAppID flag; when present, combined with the hostID and objectID parameters, shall be used to specify the GUID of the object the information carried by this VAR is about
objectID	ObjectID	Used to specify the GUID of the object the information carried by this VAR is about; if the parameters hostID and appID are present then the GUID is <VAR.hostID, VAR.appID, VAR.objectID> otherwise, combined with the modal/state values provided by the Messenger service, the GUID is <sourceHostID, sourceAppID
writerSeqNumber	SequenceNumber	Used to tag changes in the composite state provided by the CSTWriter; it is incremented each time a change in such composite state occurs; this should be a strictly positive number (1, 2, …), or the special sequence number, SEQUENCE_NUMBER_UNKNOWN, may be sent to indicate that the sender does not keep track of the sequence number
parameterSequence	ParameterSequence	Conditional to the value of the hasParameterSequence flag; it shall be used to provide a variable list of operational Issue parameters, and allows for publish/subscribe extensions

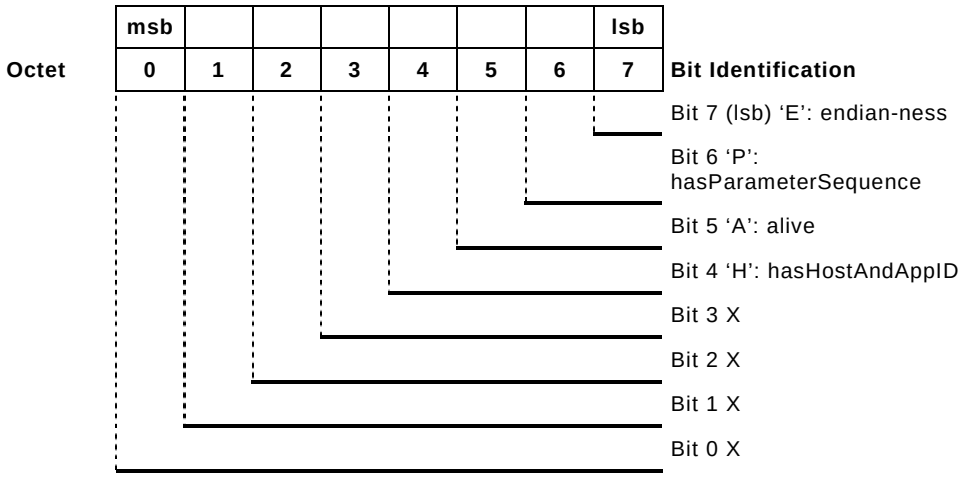

Figure 9 – Flags of VAR request

Table 50 – Meaning of VAR request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
hasParameterSequence	Specifies if there is or not a parameterSequence parameter. Values: {NO, 0}, {YES, 1}
alive	Indicates to the reader whether the data-object is alive or else is not-alive (disposed). Values: {NO, 0}, {YES, 1}
hasHostAndAppID	Specifies if there are or not the hostID and the appID parameters. Values: {NO, 0}, {YES, 1}

7.5.3.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information;

— *writerSeqNumber* is not strictly positive (1,2,...) or is SEQUENCE_NUMBER_UNKNOWN;

— the parameter sequence is invalid.

7.5.3.3 Change in state of the receiver

None.

7.5.3.4 Logical interpretation

Table 51 – Interpretation of VAR

Attribute	Value	Description
readerGUID	<destHostid, destAppId, VAR.readerObjectId>	The Reader to which the VAR applies. The VAR.readerObjectId can be OBJECTID_UNKNOWN, in which case the VAR applies to all Readers of that *writerGUID* within the Application <destHostId, destAppId>
writerGUID	<sourceHostId, sourceAppId, VAR.writerObjectId>	The CSTWriter that sent the information
objectGUID	<VAR.hostId, VAR.appId, VAR.objectId> (iff H == 1) <sourceHostId, sourceAppId, VAR.objectId> (iff H == 0)	The object this information (contained in the parameters) is about
writerSeqNumber	VAR.writerSeqNumber	Incremented each time a change in the Composite State provided by the CSTWriter occurs. This should be a strictly positive number (1, 2, …). Or, the special sequence number, SEQUENCE_NUMBER_UNKNOWN, may be sent to indicate that the sender does not keep track of the sequence number
(timestamp)	current.timestamp if current.haveTimestamp == true	(optional) This is present iff current.haveTimestamp == true. Timestamp of the new parameters sent with this sub-message
(parameters)	VAR.parameters (iff VAR.P==1)	(optional) This is present iff VAR.P == 1. Contains information about the object
ALIVE-bit	VAR.A	Indicates to the reader whether the data-object is alive or else is not-alive (disposed)
ACKIPAddressPortList	{ unicastReplyIPAddress:unicastReplyIPPort, writer>IPAddressPortList() }	Where to sent ACKs in reply to this sub-message

7.5.4 GAP FAL PDU

7.5.4.1 Request primitive

Service identifier, sub-message ID = 8 (0x08).

This sub-message is sent from a CSTWriter to a CSTReader to indicate that a range of sequence numbers is no longer relevant. The set may be a contiguous range of sequence numbers or a specific set of sequence numbers.

This is an unconfirmed service.

The format is given in Table 52, Figure 10 and Table 53.

Table 52 – GAP request

Parameter name / Field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 3 (0x03)
Flags	Octet string, 1 octet, see Figure 10 and Table 53	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
readerObjectID	ObjectID	Used to specify the reader GUID <destHostID, destAppID, GAP.readerObjectID> for which the GAP service is meant. The GAP.readerObjectID can be OBJECTID_UNKNOWN, in which case the GAP service applies to all readers within the application <destHostId, destAppId>
writerObjectID	ObjectID	Used to specify the CSTWriter GUID <sourceHostID, sourceAppID, GAP.writerObjectID> that is the subject of the GAP sequence numbers of this GAP service invocation
firstSeqNumber	SequenceNumber	Used with the *bitmap* parameter to specify the sequence numbers that are no longer available in the writerObjectID network object; the list of the no longer available sequence numbers is the union of: all the sequence numbers in the range from *GAP.firstSeqNumber* up to *GAP.bitmap.bitmapBase* – 1; this list is empty if the *firstSeqNumber* is greater than or equal to the *bitmapBase* of the *bitmap*; *GAP.firstSeqNumber* should always be greater than or equal to 1; *and* the sequence numbers that have the corresponding bit in the *bitmap* set to 1
bitmap	Bitmap	Used with the *firstSeqNumber* parameter to specify the sequence numbers that are no longer available in the writerObjectID network object, as detailed in the *firstSeqNumber* parameter description above

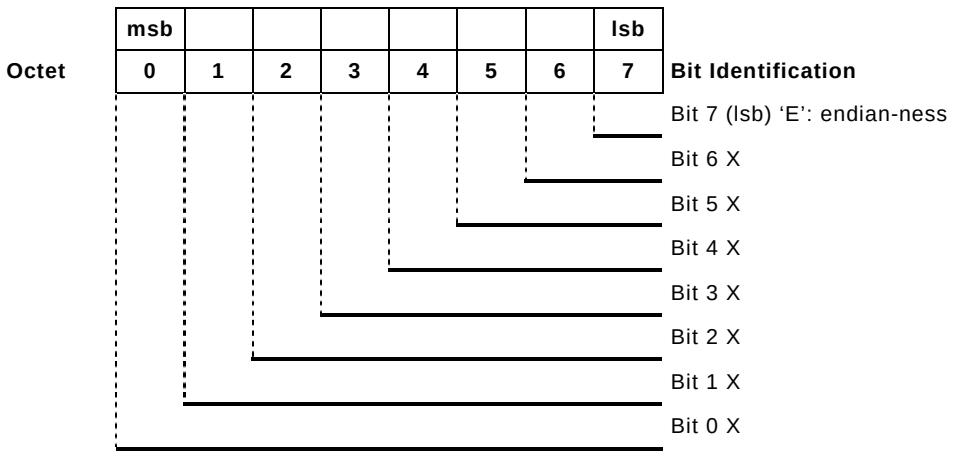

Figure 10 – Flags of GAP request

Table 53 – Meaning of GAP request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}

7.5.4.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information;

— bitmap is invalid;

— *firstSeqNumber* is 0 or negative.

7.5.4.3 Change in state of the receiver

None.

7.5.4.4 Logical interpretation

Table 54 – Interpretation of GAP

Attribute	Value	Description
readerGUID	<destHostId, destAppId, GAP.readerObjectId>	The GUID of the CSTReader for which the *gapList* is meant. The GAP.readerObjectId can be OBJECTID_UNKNOWN, in which case the GAP applies to all Readers within the Application <destHostId, destAppId>
writerGUID	<sourceHostId, sourceAppId, GAP.writerObjectId>	The GUID of the CSTWriter to which the *gapList* applies
ACKIPAddressPortList	{ unicastReplyIPAddress:unicastReplyPort }	If the CSTReader that receives this sub-message needs to reply with an ACK sub-message, then this ACK can be sent to one of the explicit destinations in this list
gapList	{ GAP.firstSeqNumber, GAP.firstSeqNumber+1,..., GAP.bitmap.bitmapBase-1 } *and* all sequence numbers that have a corresponding bit set to 1 in the bitmap	The list of sequence numbers that are no longer available in the *writerObject*. This list is the union of: All the sequence numbers in the range from *GAP.firstSeqNumber* up to *GAP.bitmap.bitmapBase* - 1. This list is empty if the *firstSeqNumber* is greater than or equal to the *bitmapBase* of the *bitmap*. *GAP.firstSeqNumber* should always be greater than or equal to 1; *and* The sequence numbers that have the corresponding bit in the *bitmap* set to 1

7.5.5 ACK FAL PDU

7.5.5.1 Request primitive

Service identifier, sub-message ID = 6 (0x06).

This sub-message is used to communicate the state of a Reader to a Writer.

This is an unconfirmed service.

The format is given in Table 55, Figure 11 and Table 56.

Table 55 – ACK request

Parameter name / Field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 3 (0x03)
Flags	Octet string, 1 octet, see Figure 11 and Table 56	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
readerObjectID	ObjectID	Used to specify the GUID <sourceHostID, sourceAppID, ACK.readerObjectID> of the reader acknowledges receipt of certain sequence numbers and/or requests to receive certain sequence numbers
writerObjectID	ObjectID	Used to specify the GUID <destHostID, deatAppID, ACK.writerObjectID> of the writer that the reader has received these sequence numbers from and/or wants to receive these sequence numbers from
bitmap	Bitmap	Used to specify the ACK botmap: a “0” in this bitmap means that the corresponding sequence-number is missing; a “1” in the bitmap conveys no information, that is, the corresponding sequence number may or may not be missing; by sending an ACK, the readerGUID object acknowledges receipt of all messages up to and including the sequence number (bitmap.bitmapBase -1)

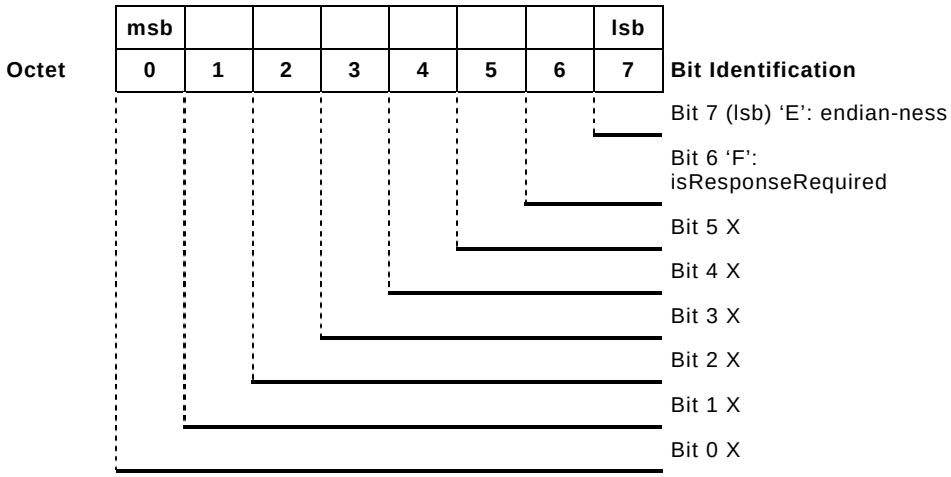

Figure 11 – Flags of ACK request

Table 56 – Meaning of ACK request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
responseIsNotRequired	Specifies if the application sending the Heartbeat requires or not a response. Known with 'F', that stands for 'Final'. Values: {FALSE, it IS required, 0}, {TRUE, it IS NOT required, 1}

7.5.5.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information;

— bitmap is invalid.

7.5.5.3 Change in state of the receiver

None.

7.5.5.4 Logical interpretation

Table 57 – Interpretation of ACK

Attribute	Value	Description
FINAL-bit	ACK.F	When the F-bit is set, the application sending the ACK does not expect a response to the ACK
readerGUID	<sourceHostId, sourceAppId, ACK.readerObjectId>	The GUID of the Reader that acknowledges receipt of certain sequence numbers and/or requests to receive certain sequence numbers
writerGUID	<destHostId, destAppId, ACK.writerObjectId>	The GUID of the Writer that the reader has received these sequence numbers from and/or wants to receive these sequence numbers from
replyIPAddressPortList	{ unicastReplyIPAddress:unicastReplyPort, multicastReplyIPAddress: multicastReplyPort }	This is an additional list of addresses that the receiving application can use to respond to this ACK
bitmap	ACK.bitmap	A "0" in this bitmap means that the corresponding sequence-number is missing. A "1" in the bitmap conveys no information, that is, the corresponding sequence number may or may not be missing. By sending an ACK, the readerGUID object acknowledges receipt of all messages up to and including the sequence number (bitmap.bitmapBase -1)

7.5.6 Header FAL PDU

7.5.6.1 General

It is convenient to see the message header as a sub-APDU itself, despite it being a header. In IEC 61158-5-15 it has been abstracted as one of the services of the messenger ASE because

it has many similarities with other services there, which modify defaults set by the header. It is also different, its sub-APDU has the fixed length of 16 octets, it is a singleton and it is identified by its location instead of having a service identifier.

The header is one of the set of logistic sub-messages: header, INFO_DST, INFO_REPLY, INFO_SRC, INFO_TS, and PAD.

7.5.6.2 Request primitive

Service identifier: positional, it is at the beginning of every message.

The format is given in Table 58.

Table 58 – Header request

Parameter name / field	Type	Description
1st header marker	Octer string, 1 octet	Ascii character 'R'.
2nd header marker	Octet string, 1 octet	Ascii character 'T'.
3rd header marker	Octet string, 1 octet	Ascii character 'P'.
4th header marker	Octet string, 1 octet	Ascii character 'S'.
version	ProtocolVersion	The type is defined in IEC 61158-5-15. This specification refers to {major = 0x1, minor = 0x0}
vendorID	VendorID	Used to specify the vendor of the middleware implementing the publish/subscribe protocol and allows this vendor to add specific extensions to the protocol. The vendorID does not refer to the vendor of the device or product that contains publish/subscribe middleware. The type is defined in IEC 61158-5-15
hostID	HostID	Used to to specify sourceHostID for the receiver and for the reader and writer services that use sourceHostID; the value setting is modal, as specified until changed
appID	AppID	Used to to specify sourceAppID for the receiver and for the reader and writer services that use sourceAppID; the value setting is modal, as specified until changed

7.5.6.3 Validity

A header is *invalid* when any of the following are true:

— The full APDU has less than the required number of octets to contain a full header (16 octets);

— its first 4 octets are not 'R' 'T' 'P' 'S';

— the major protocol version is larger than the major protocol version supported by the implementation.

7.5.6.4 Change in state of the receiver

Table 59 – Change in state of the receiver

attribute	value
sourceHostID	Header.hostID
sourceAppID	Header.appID
sourceVersion	Header.version
sourceVendorID	Header.vendorID
haveTimestamp	false

7.5.6.5 Logical interpretation

None.

7.5.7 INFO_DST FAL PDU

7.5.7.1 Request primitive

Service identifier, sub-message ID = 14 (0x0E).

This sub-message modifies the logical destination for the service requests that follow it.

This is an unconfirmed service.

The format is given in Table 60, Figure 12 and Table 61.

Table 60 – INFO_DST request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 14 (0x0E)
Flags	Octet string, 1 octet, see Figure 12 and Table 61	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
hostID	HostID	Used to specify destHostID for the reader and writer services that use destHostID; the value setting is modal, as specified until changed
appID	AppID	Used to specify destAppID for the reader and writer services that use destAppID; the value setting is modal, as specified until changed

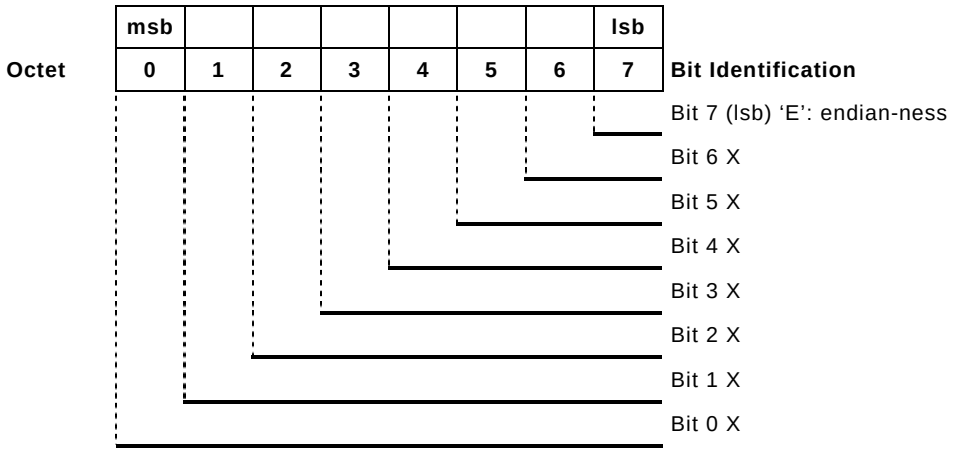

Figure 12 – Flags of INFO_DST request

Table 61 – Meaning of INFO_DST request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}

7.5.7.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information.

7.5.7.3 Change in state of the receiver

```
if(INFO_DST.hostId != HOSTID_UNKNOWN) {
    destHostId = INFO_DST.hostId
} else {
    destHostId = hostId of application receiving the message
}

if(INFO_DST.appId != APPID_UNKNOWN) {
    destAppId = INFO_DST.appId
} else {
    destAppId = appId of application receiving the message
}
```

In other words, an INFO_DST with a HOSTID_UNKNOWN means that any host may interpret the following submessages as if they were meant for it. Similarly, an INFO_DST with a APPID_UNKNOWN means that any application may interpret the following submessages as if they were meant for it.

7.5.7.4 Logical interpretation

None; this only affects the interpretation of the submessages that follow it.

7.5.8 INFO_REPLY FAL PDU

7.5.8.1 Request primitive

Service identifier, sub-message ID = 13 (0x0D).

This sub-message specifies explicit information on where to send a reply to the requests that follow it within the same message.

This is an unconfirmed service.

The format is given in Table 62, Figure 13 and Table 63.

Table 62 – INFO_REPLY request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 13 (0x0D)
Flags	Octet string, 1 octet, see Figure 13 and Table 63	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
unicastReplyIPAddress	IPAddress	Used to specify the IP address of where to send a reply to the requests that follow within the same message; the value setting is modal, as specified until changed
unicastReplyPort	Port	Used to specify the port of where to send a reply to the requests that follow within the same message; the value setting is modal, as specified until changed
multicastReplyIPAddress	IPAddress	Conditional to the value of the hasMulticast parameter. It shall be used to specify the IP address of where to send a reply to the requests that follow within the same message; the value setting is modal, as specified until changed
multicastReplyPort	Port	Conditional to the value of the hasMulticast parameter. It shall be used to specify the port of where to send a reply to the requests that follow within the same message; the value setting is modal, as specified until changed

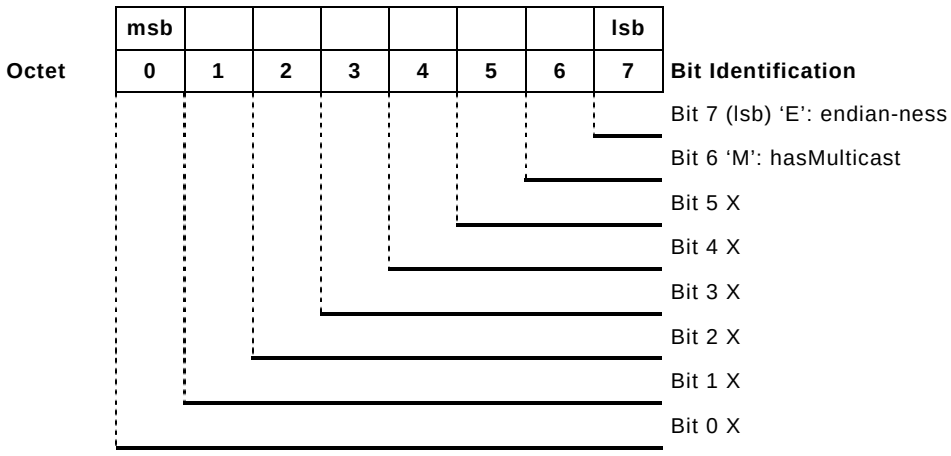

Figure 13 – Flags of INFO_REPLY request

Table 63 – Meaning of INFO_REPLY request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
hasMulticast	Specifies if there is multicast information. Values: {NO, 0}, {YES, 1}

7.5.8.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information.

7.5.8.3 Change in state of the receiver

```
if ( INFO_REPLY.unicastReplyIPAddress != IPADDRESS_INVALID) {
    unicastReplyIPAddress  = INFO_REPLY.unicastReplyIPAddress;
}
unicastReplyPort  = INFO_REPLY.replyPort
if ( M==1 ) {
    multicastReplyIPAddress  = INFO_REPLY.multicastReplyIPAddress
    multicastReplyPort       = INFO_REPLY.multicastReplyPort
} else {
    multicastReplyIPAddress  = IPADDRESS_INVALID
    multicastReplyPort       = PORT_INVALID
}
```

7.5.8.4 Logical interpretation

None; this only affects the interpretation of the submessages that follow it.

7.5.9 INFO_SRC FAL PDU

7.5.9.1 Request primitive

Service identifier, sub-message ID = 12 (0x0C).

This sub-message modifies the logical source for the service requests that follow it.

This is an unconfirmed service.

The format is given in Table 64, Figure 14 and Table 65.

Table 64 – INFO_SRC request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 12 (0x0C)
Flags	Octet string, 1 octet, see Figure 14 and Table 65	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
appIPAddress	IPAddress	Used to specify the IP address of where to send a reply to the requests that follow within the same message; the value setting is modal, as specified until changed
version	ProtocolVersion	The type is defined in IEC 61158-5-15. This specification refers to {major = 0x1, minor = 0x0}
vendorID	VendorID	Used to specify the vendor of the middleware implementing the publish/subscribe protocol and allows this vendor to add specific extensions to the protocol. The vendorID does not refer to the vendor of the device or product that contains publish/subscribe middleware. The type is defined in IEC 61158-5-15
hostID	HostID	Used to specify sourceHostID for the receiver and for the reader and writer services that use sourceHostID; the value setting is modal, as specified until changed
appID	AppID	Used to specify sourceAppID for the receiver and for the reader and writer services that use sourceAppID; the value setting is modal, as specified until changed

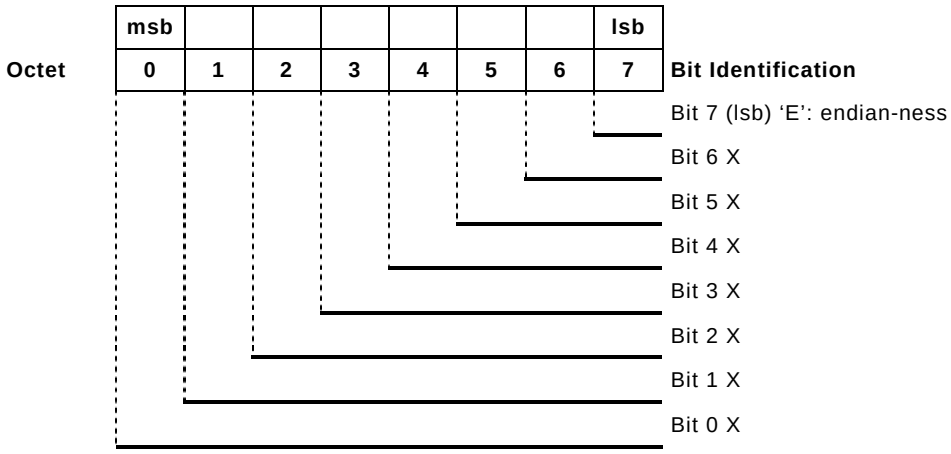

Figure 14 – Flags of INFO_SRC request

Table 65 – Meaning of INFO_SRC request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}

7.5.9.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information.

7.5.9.3 Change in state of the receiver

```
sourceHostId             = INFO_SRC.hostId
sourceAppId              = INFO_SRC.appId
sourceVersion            = INFO_SRC.version
sourceVendorId           = INFO_SRC.vendorId
unicastReplyIPAddress    = INFO_SRC.appIPAddress
unicastReplyPort         = PORT_INVALID
multicastReplyIPAddress  = IPADDRESS_INVALID
multicastReplyPort       = PORT_INVALID
haveTimestamp            = false
```

7.5.9.4 Logical interpretation

None; this only affects the interpretation of the submessages that follow it.

7.5.10 INFO_TS FAL PDU

7.5.10.1 Request primitive

Service identifier, sub-message ID = 9 (0x09).

This sub-message is used to send a timestamp which applies to the service requests that follow it.

This is an unconfirmed service.

The format is given in Table 66, Figure 15 and Table 67.

Table 66 – INFO_TS request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 9 (0x09)
Flags	Octet string, 1 octet, see Figure 15 and Table 67	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF
ntpTimestamp	NtpTime	Conditional to the value of the noTimestamp flag. It shall be used to specify the timestamp for the requests that follow within the same message; the value setting is modal, as specified until changed

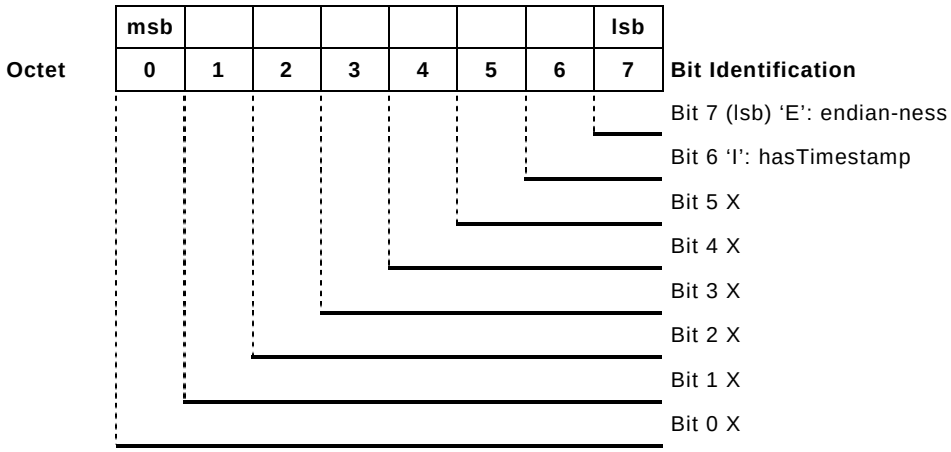

Figure 15 – Flags of INFO_TS request

Table 67 – Meaning of INFO_TS request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}
hasTimestamp	Specifies if there is (0) or not (1) a timestamp. Values: {hasTimestamp, 0}, {thereIsNOTimestamp, 1}

7.5.10.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information.

7.5.10.3 Change in state of the receiver

```
if (INFO_TS.I==0) {
    haveTimestamp   =  true
    timestamp       = INFO_TS.ntpTimestamp
} else {
    haveTimestamp   =  false
}
```

7.5.10.4 Logical interpretation

None; this only affects the interpretation of the submessages that follow it.

7.5.11 PAD FAL PDU

7.5.11.1 Request primitive

Service identifier, sub-message ID = 1 (0x01).

This sub-message is used for alignment purposes within the message, whereas the receiver will skip the PAD service request length and will get to the next request. The length is specified as part of the request format. The service has no other meaning.

This is an unconfirmed service.

The format is given in Table 68, Figure 16 and Table 69.

Table 68 – PAD request

Parameter name / field	Type	Description
Sub-message ID	Unsigned8	Service identifier, sub-message ID = 1 (0x01)
Flags	Octet string, 1 octet, see Figure 16 and Table 69	Octet encoding boolean flags, described separately
OctetsToNextHeader	Unsigned16	Number of octets from the first octet of the contents of this sub-message until the first octet of the header of the next sub-message, for all sub-messages but the last. See 7.3.4. Allowed values: 0x0000 to 0xFFFF

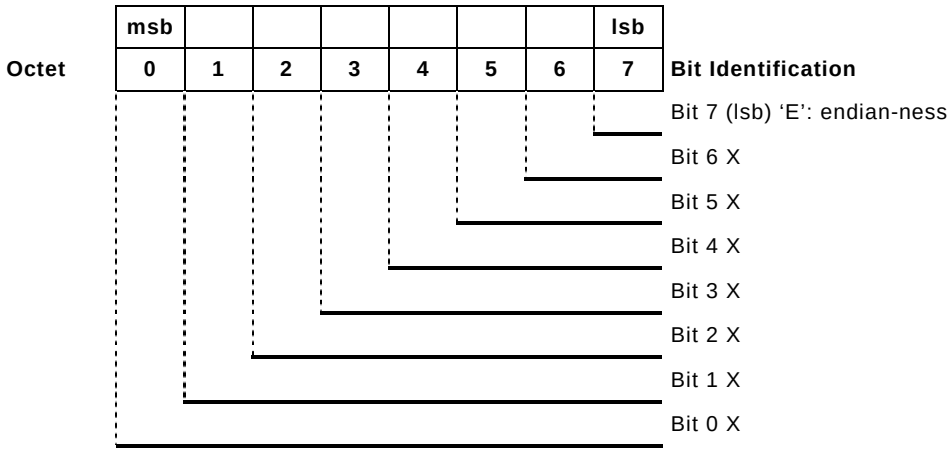

Figure 16 – Flags of PAD request

Table 69 – Meaning of PAD request flags

Boolean flag	Description
endian-ness	Specifies the endian-ness of this service request and indication. Values: {big-endian, 0}, {little-endian, 1}

7.5.11.2 Validity

This sub-message is *invalid* when any of the following are true:

— *octetsToNextHeader* is too small given other sub-messages and message information.

7.5.11.3 Change in state of the receiver

None.

7.5.11.4 Logical interpretation

Logistic only, the receiver skips the PAD using *octetsToNextHeader*.

7.6 Common data representation for publish/subscribe

7.6.1 General

The following is a summary of the CDR (defined within the OMG CORBA protocol) data format and the OMG IDL syntax to the extent that they are used by the publish/subscribe protocol and its description in this document.

NOTE The authoritative source of the CDR specification and OMG IDL is the CORBA protocol (available through the Object Management Group <http://www.omg.org>). In the CORBA V2.3.1 specification, the relevant sections are 15.3 (General Inter-ORB Protocol—CDR Transfer Syntax) and 3.10 (OMG IDL Syntax and Semantics— Type Declaration). Unless mentioned explicitly, CDR for publish/subscribe follows the CDR standard for GIOP version 1.1.

publish/subscribe makes some additional restrictions on CDR and makes concrete choices where CDR for GIOP 1.1 is not fully defined. Notable are the implementation of the wide characters and strings (wchar and wstring) and the definition of the publish/subscribe Identifier, which only allows certain characters.

7.6.2 Primitive Types

7.6.2.1 Semantics

Table 70 – Semantics

OMG IDL-name	Size	Meaning
octet	1	8 uninterpreted bits
boolean	1	TRUE or FALSE
unsigned short	2	integer N, 0 <= N < 2^16
short	2	integer N, -2^15 <= N < 2^15
unsigned long	4	integer N, 0 <= N < 2^32
long	4	integer N, -2^31 <= N < 2^31
unsigned long long	8	integer N, 0 <= N < 2^64
long long	8	integer N, -2^63 <= N < 2^63
float	4	IEEE single-precision fp number
double	8	IEEE double-precision fp number
char	1	a character following ISO8859-1
wchar	2	a wide-character following UNICODE

Remarks:

— CDR defines some additional primitive types, such as "long double"; these are currently disallowed by publish/subscribe;

— CDR leaves the width of the wchar open; publish/subscribe gives it a fixed length of two octets.

7.6.2.2 Encoding

CDR has both a big-endian ("BE") and a little-endian ("LE") encoding. The sender is allowed to choose the encoding. The receiver needs to know which encoding has been used by the sender to unpack the data correctly. This endianness-bit is transmitted as part of the publish/subscribe protocol.

7.6.2.3 octet

An octet is encoded as shown in Figure 17.

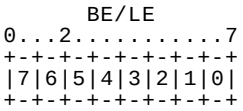

Figure 17 – Encoding of octet

7.6.2.4 boolean

A boolean is encoded as shown in Figure 18.

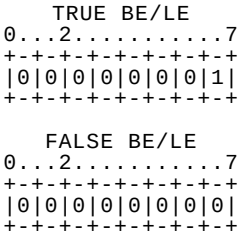

Figure 18 – Encoding of boolean

7.6.2.5 unsigned short

An unsigned short is encoded as shown in Figure 19.

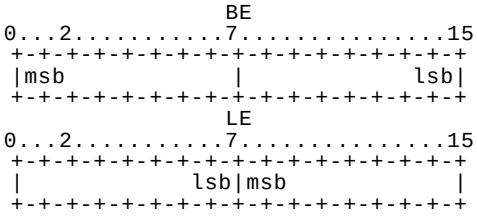

Figure 19 – Encoding of unsigned short

7.6.2.6 short

A short has the same encoding as an unsigned short, but uses 2's complement representation.

7.6.2.7 unsigned long

An unsigned long is encoded as shown in Figure 20.

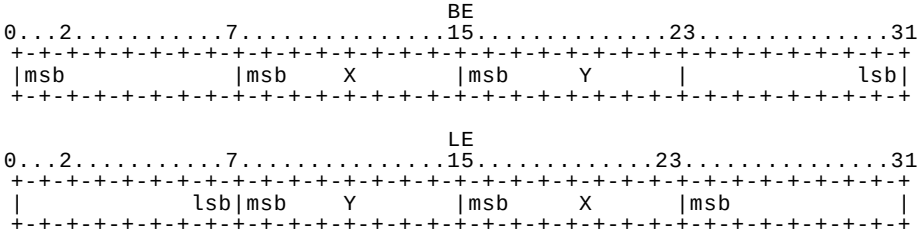

Figure 20 – Encoding of unsigned long

7.6.2.8 long

A long has the same encoding as an unsigned long, but uses 2's complement representation.

7.6.2.9 unsigned long long

An unsigned long long is encoded as in Figure 21.

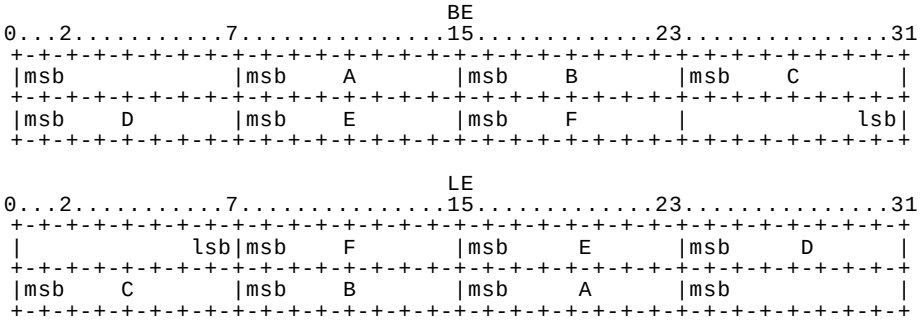

Figure 21 – Encoding of unsigned long long

7.6.2.10 long long

A long long has the same encoding as an unsigned long long, but uses 2's complement representation.

7.6.2.11 float

A float is encoded as shown in Figure 22.

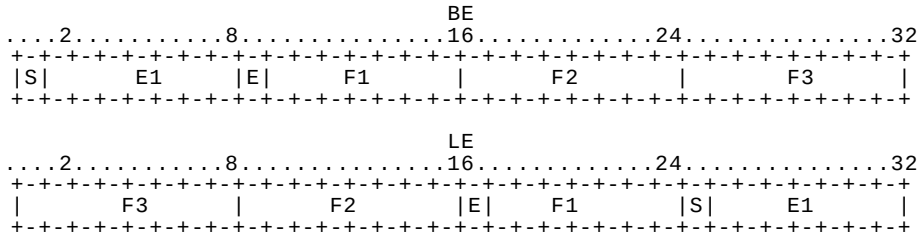

Figure 22 – Encoding of float

7.6.2.12 double

A double is encoded as shown in Figure 23.

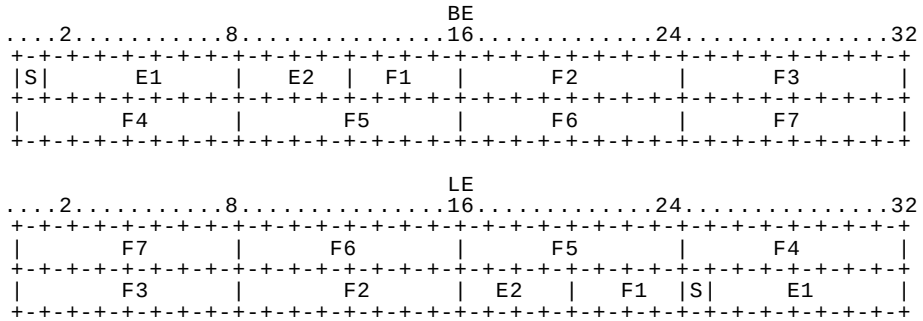

Figure 23 – Encoding of double

7.6.2.13 char

A character has the same encoding as an octet.

7.6.2.14 wchar

A wide-character occupies two octets and follows UNICODE encoding.

7.6.3 Constructed types

7.6.3.1 Alignment

In CDR, only the primitive types listed in 7.6.2 have alignment constraints. The primitive types need to be aligned on their length. For example, a long must start on a 4-octets boundary. The boundaries are counted from the start of the CDR stream.

7.6.3.2 Identifiers

An identifier is a sequence of ASCII alphabetic and numeric characters, plus the underscore character. The first character must be an ASCII alphabetic character.

7.6.3.3 List of constructed types

publish/subscribe supports the following subset of CDR constructed types:

struct	structure
array	fixed size array (the length is part of the type)
sequence	variable size array (the maximum length is part of the type)
string	string of 1-byte characters
wstring	string of wide character

NOTE There are some additional constructed types in CDR, such as unions and fixed-point decimal types; these are currently not supported in publish/subscribe.

7.6.3.4 Struct

A structure has a name (an identifier) and an ordered sequence of elements. Each element has a name (an identifier) and a type. In OMG IDL, a structure is defined by the keyword "struct", followed by an identifier and a sequence of the elements of the structure. An example of the definition of a structure named "myStructure" in OMG IDL is:

```
struct myStructure {
    long long l;
    unsigned short s;
    myType t;
}
```

In CDR, the components of such a structure are encoded in the order of their declaration in the structure. The only alignment requirements are at the level of the primitive types.

7.6.3.5 Enumeration

An enumeration has a name (an identifier) and an ordered set of case-keywords which also are identifiers. In OMG IDL, an enumeration is defined by the keyword "enum", followed by an identifier and a list of identifiers in the enumeration. For example:

```
enum myEnumeration { case1, case2, case3 }
```

In CDR, enumerations are encoded as unsigned longs, where the identifiers in the enumeration are numbered from left to right, starting with 0.

7.6.3.6 Sequence

A sequence is a variable number of elements of the same type. Optionally, the type can specify the maximum number of elements in the sequence. OMG IDL uses the keyword "sequence". The syntax for an unbounded sequence of floats is:

```
sequence<float>
```

The syntax for a sequence of unsigned long longs with a maximum length is:

```
sequence<unsigned long long, MAX_NUMBER_OF_ELEMENTS>
```

In CDR, sequences are encoded as the number of elements (as an unsigned long) followed by each of the elements in the sequence.

7.6.3.7 Array

Arrays have a fixed and well-known number of elements of the same type. In OMG IDL, an array is defined using the symbols "[" and "]", following the C/C++ style. An example is:

```
float[17]
```

In CDR, arrays are encoded by encoding each of its elements from low to high index. In multi-dimensional arrays, the index of the last dimension varies most quickly.

7.6.3.8 String

A string is an optionally bounded sequence of characters. In OMG IDL, a string of unbounded length is identified by the keyword "string"; a bounded string is specified as follows:

```
string<MAX_LENGTH>
```

On the wire, strings are encoded as an unsigned long (indicating the number of octets that follow to encode the string), followed by each of the characters in the string and a terminating zero. For example, the string "Hello" is encoded as the unsigned long 6 followed by the octets 'H', 'e', 'l', 'l', 'o', 0.

7.6.3.9 Wstring

A wide-string is a string of wide-characters. In OMG IDL, unbounded and bounded strings are specified, respectively, as follows:

```
wstring
wstring<MAX_LENGTH>
```

In CDR (GIOP 1.1), a wide-string is encoded as an unsigned long indicating the length of the string on octets or unsigned integers (determined by the transfer syntax for wchar), followed by the individual wide characters. Both the string length and contents include a terminating NULL.

8 Structure of FAL protocol state machines

Interface to FAL services and protocol machines are specified in this subclause.

NOTE The state machines specified in this subclause and ARPMs defined in the following sections only define the valid events for each. It is a local matter to handle the invalid events.

The behavior of the FAL is described by three integrated protocol machines. Specific sets of these protocol machines are defined for different AREP types. The three protocol machines are: FAL Service Protocol Machine (FSPM), the Application Relationship Protocol Machine (ARPM), and the data-link layer Mapping Protocol Machine (DMPM). The relationship among these protocol machines as well as primitives exchanged among them are depicted in Figure 24.

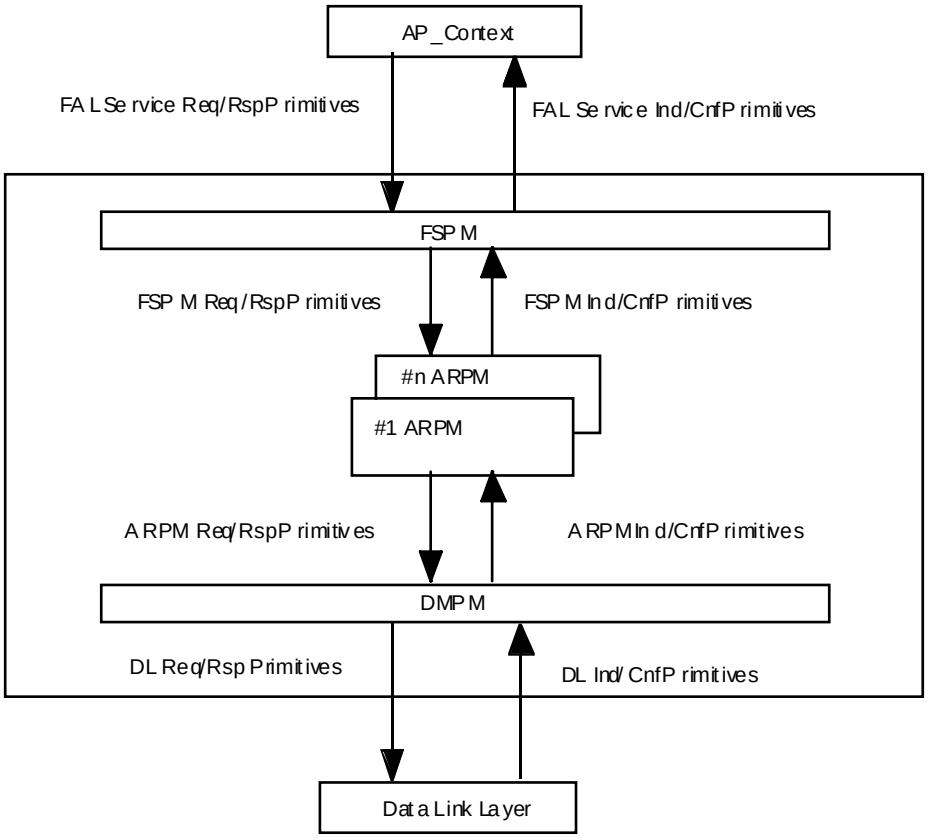

Figure 24 – Relationships among protocol machines and adjacent layers

The FSPM describes the service interface between the AP-Context and a particular AREP. The FSPM is common to all the AREP classes and does not have any state changes. The FSPM is responsible for the following activities:

a) to accept service primitives from the FAL service user and convert them into FAL internal primitives;

b) to select an appropriate ARPM state machine based on the AREP Identifier parameter supplied by the AP-Context and send FAL internal primitives to the selected ARPM;

c) to accept FAL internal primitives from the ARPM and convert them into service primitives for the AP-Context;

d) to deliver the FAL service primitives to the AP-Context based on the AREP Identifier parameter associated with the primitives.

The ARPM describes the establishment and release of an AR and exchange of FAL-PDUs with a remote ARPM(s). The ARPM is responsible for the following activities:

a) to accept FAL internal primitives from the FSPM and create and send other FAL internal primitives to either the FSPM or the DMPM, based on the AREP and primitive types;

b) to accept FAL internal primitives from the DMPM and send them to the FSPM as a form of FAL internal primitives;

c) if the primitives are for the Establish or Abort service, it shall try to establish or release the specified AR.

The DMPM describes the mapping between the FAL and the DLL. It is common to all the AREP types and does not have any state changes. The DMPM is responsible for the following activities:

d) to accept FAL internal primitives from the ARPM, prepare DLL service primitives, and send them to the DLL;

e) to receive DLL indication or confirmation primitives from the DLL and send them to the ARPM in a form of FAL internal primitives.

9 AP-context state machines for client/server

There is no AP-Context State Machine in this Type of FAL.

10 FAL service protocol machine (FSPM) for client/server

10.1 General

FAL Service Protocol Machine is common to all the AREP types. Only applicable primitives are different among different AREP types. It has one state called "ACTIVE" as shown in Figure 25.

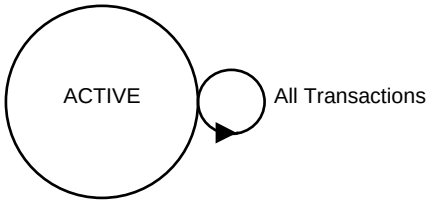

Figure 25 – State transition diagram of FSPM

10.2 FSPM state tables

Table 71 and Table 72 specifiy the FSPM protocol machine.

The invoke_ID used in the protocol machine is the transaction identifier object and it must be unique across all the transaction identifiers still pending on the connection involved.

The transaction object is instantiated for the duration of the service invocation, and in general it is used to couple requests and confirmations for confirmed services, and destroyed afterward. The service invocation is supposed to be completed within a device and connection specific amount of time. If the prescribed amount of time expires, then the transaction object is discarded as the confirmation matching the associated request is no longer expected, and the client is notified.

The transaction object also acts as a moderator. If a client can only instantiate one transaction object at a time then, when invoking a service, it is not necessary to dynamically create a transaction object and exchange its identifier with lower layers, since the main reason for having a transaction object is to properly couple requests with confirmations, which is automatic for these clients. In these situations a single static transaction object effectively acts like a client token, which is either taken or available, and only of interest to the client. For these clients there will be only one pending request at a time, and the invoke_ID as used in the state tables may be considered empty in the request and always matching in the confirmation.

For unconfirmed services, transaction objects are disposed-of after an amount of time that is device and connection specific. This allows for proper propagation.

On a given connection, a new transaction can only take place if the request for the transaction object is granted. Therefore, all the .req events in the FSPM client transactions state table reported in Table 71 have to be interpreted as being produced after the client successfully obtains a transaction object for the particular instantiation of the .req event.

There is a transaction state machine associated with every connection. The transaction state machine for a connection is illustrated in Figure 26. It has one state called "ACTIVE".

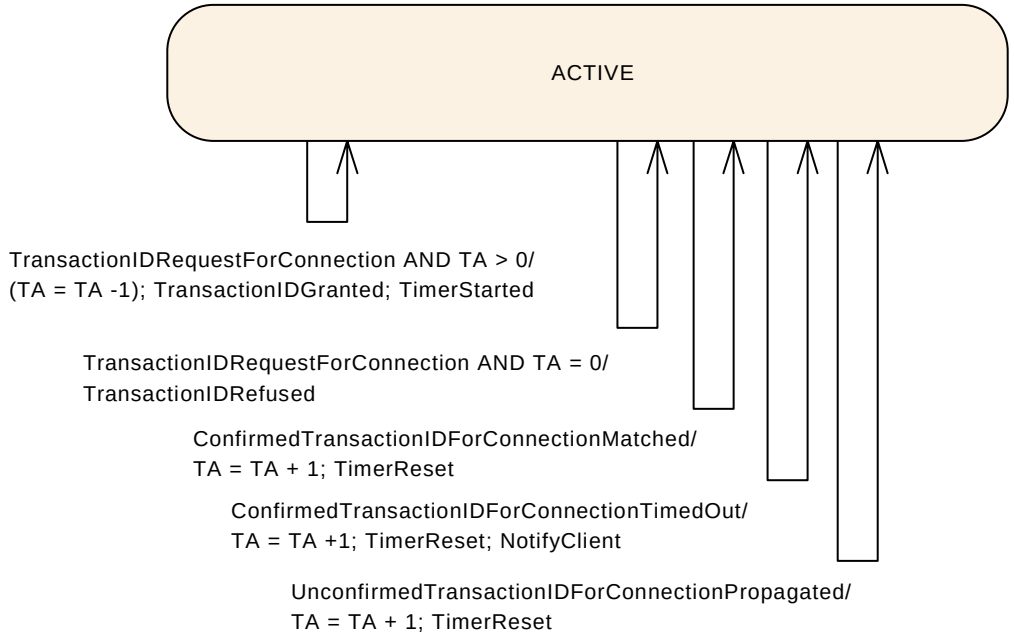

Figure 26 – Transaction state machine, per connection

In Figure 26, the TransactionIDRequestForConnection is the event used by the client to request a new transaction object, and the associated Invoke_ID, for a given connection.

TA is the number of transactions that are currently available. The initial setting of the TA parameter is implementation and connection dependent, and it may be per connection or across connections. It is in general related to the type of connection and to platform resources.

The ConfirmedTransactionIDForConnectionMatched is a side event generated by the FSPM client transactions state machine when, upon receipt of a .cnf event, the function MatchInvokeID(Invoke_ID) is successful.

There is one timer per transaction object.

The ConfirmedTransactionIDForConnectionTimedOut is an event generated by the transaction object associated timer. Beside local state machine maintenance, this event produces a client notification.

The UnconfirmedTransactionIDForConnectionPropagated is an event generated by the transaction object associated timer, used to self-pace the transaction activity after an unconfirmed transaction, if needed.

When upon reception of a .cnf event the MatchInvokeID(Invoke_ID) fails, either because the transaction object associated timer expired and that Invoke_ID was no longer expected, or due of other reasons, the associated confirmation is discarded.

Table 71 – FSPM state table – client transactions

#	Current state	**Event** or **condition** **=> action**	Next state

S1	ACTIVE	ReadDiscretes.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S2	ACTIVE	ReadCoils.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S3	ACTIVE	WriteSingleCoil.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S4	ACTIVE	WriteMultipleCoils.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S5	ACTIVE	BroadcastWriteSingleCoil.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S6	ACTIVE	BroadcastWriteMultipleCoils.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S7	ACTIVE	ReadInputRegisters.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S8	ACTIVE	ReadHoldingRegisters.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S9	ACTIVE	WriteSingleHoldingRegister.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S10	ACTIVE	WriteMultipleHoldingRegisters.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

Table 71 *(continued)*

#	Current state	**Event** or **condition** **=> action**	Next state

S11	ACTIVE	MaskWriteHoldingRegister.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S12	ACTIVE	Read/WriteHoldingRegisters.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S13	ACTIVE	ReadFIFO.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S14	ACTIVE	BroadcastWriteSingleHoldingRegister.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S15	ACTIVE	BroadcastWriteMultipleHoldingRegisters.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S16	ACTIVE	ReadFileRecord.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S17	ACTIVE	WriteFileRecord.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S18	ACTIVE	ReadDeviceIdentification.req
		=>
		Transaction.req {
		remote_address := AREP_ID,
		invoke_ID := unique_per_connection_ID,
		user_data := alpdu
		}
			ACTIVE

S19	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadDiscretes.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S20	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadCoils.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

Table 71 *(continued)*

#	Current state	**Event** or **condition** **=> action**	Next state

S21	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		WriteSingleCoil.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S22	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		WriteMultipleCoils.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S23	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadInputRegisters.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S24	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadHoldingRegisters.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S25	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		WriteSingleHoldingRegister.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S26	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		WriteMultipleHoldingRegister.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S27	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		MaskWriteHoldingRegister.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S28	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		Read/WriteHoldingRegisters.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S29	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadFIFO.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

Table 71 *(continued)*

#	Current state	**Event** or **condition** **=> action**	Next state

S30	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadFileRecord.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S31	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		WriteFileRecord.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S32	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “True”
		=>
		ReadDeviceIdentification.cnf(-) {
		ArepID := remote_address,
		ExceptionCode := data
		}
			ACTIVE

S33	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadDiscretes.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S34	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadCoils.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S35	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		WriteSingleCoil.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S36	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		WriteMultipleCoils.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S37	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadInputRegisters.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S38	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadHoldingRegisters.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

Table 71 *(continued)*

#	Current state	**Event** or **condition** **=> action**	Next state

S39	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		WriteSingleHoldingRegister.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S40	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		WriteMultipleHoldingRegister.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S41	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		MaskWriteHoldingRegister.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S42	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		Read/WriteHoldingRegisters.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S43	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadFIFO.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S44	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadFileRecord.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S45	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		WriteFileRecord.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

S46	ACTIVE	Transaction.cnf
		&& MatchInvokeID(Invoke_ID)
		&& HighBit(code) == “False”
		=>
		ReadDeviceIdentification.cnf(+) {
		ArepID := remote_address,
		ResponseData := data
		}
			ACTIVE

Table 72 – FSPM state table – server transactions

#	Current state	**Event** or **condition** **=> action**	Next state

R1	ACTIVE	Transaction.ind
		=>
		al_service.ind {
		Invoke_Id := invoke_id,
		Arep_Id := arep_id,
		alpdu := user_data
		}
			ACTIVE

R2	ACTIVE	Transaction.rsp
		=>
		al_service.rsp {
		Invoke_Id := invoke_id,
		Arep_Id := arep_id,
		alpdu := user_data
		}
			ACTIVE

NOTE al_service includes all AL services. The appropriate indication or response primitive is used depending upon the type of service.			

10.3 Functions used by FSPM

Table 73 and Table 74 define the functions used by the FSPM

Table 73 – Function MatchInvokeID()

Name	MatchInvokeID	Used in	FSPM
Input		Output	
Invoke_ID		True || False	
Function			
Matches requests with responses, and upon successful match disposed of the transaction object.			

Table 74 – Function HighBit()

Name	HighBit	Used in	FSPM
Input		Output	
Code		True || False	
Function			
Checks the code field to see if it is an exception (msb of the octet is high)			

10.4 Parameters of FSPM/ARPM primitives

The parameters used with the primitives exchanged between the FSPM and the ARPM are described in Table 75.

Table 75 – Parameters used with primitives exchanged between FSPM and ARPM

Parameter name	Description
Invoke_ID	This parameter conveys the value used to match requests and responses.
arep_id	This parameter is used to identify the instance of the AREP that has issued a primitive. A means for such identification is not specified by this specification.
code	code field of the APDU.
user_data	This parameter conveys user data.

10.5 Client/server server transactions

10.5.1 General

The server transactions are detailed in Figure 27.

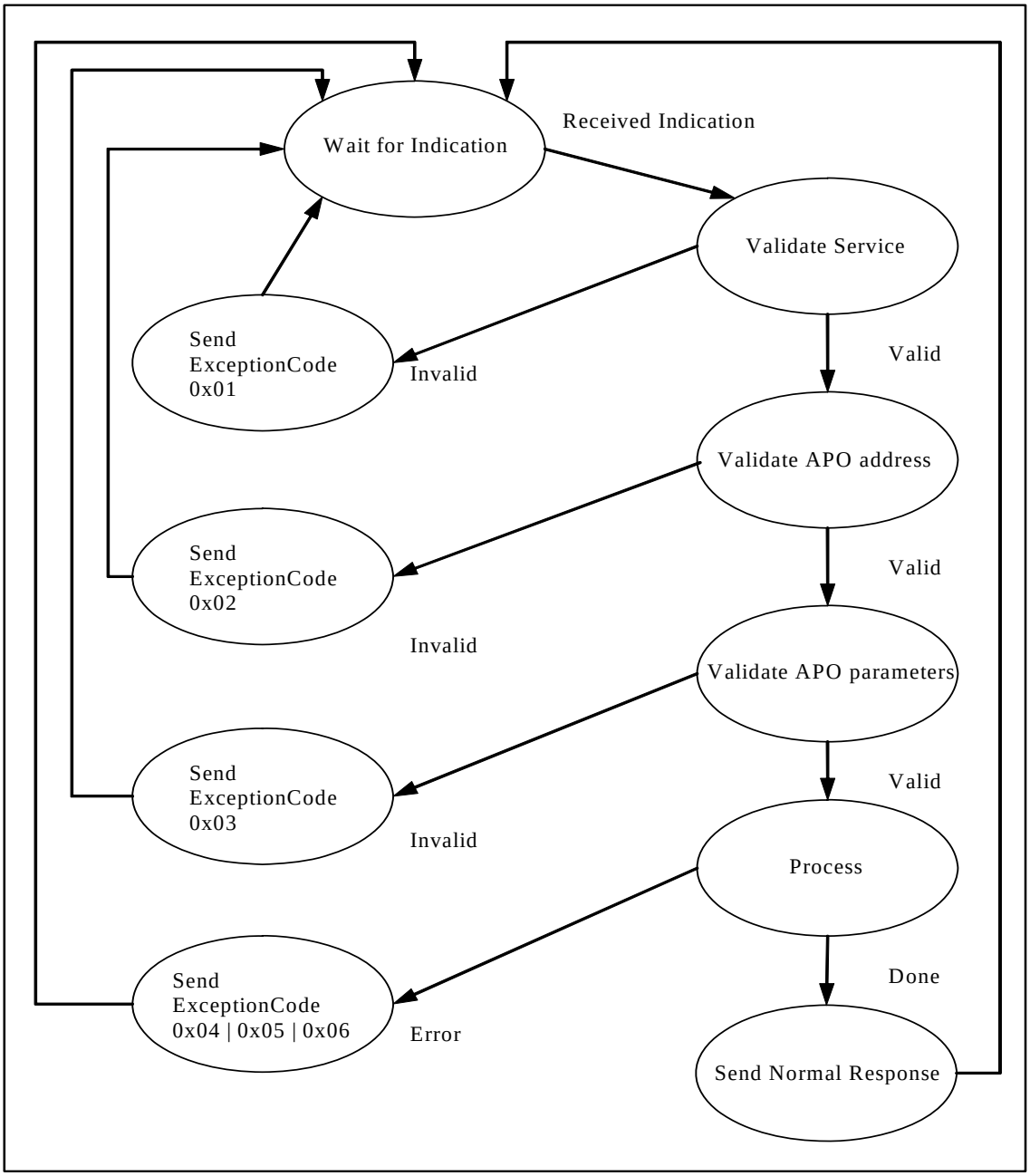

Figure 27 – Client/server server transactions

The client is responsible for the parsing of the confirmation beyond the reception of a negative confirmation encoded via exception codes. While there is no standard encoding, the following two cases must be handled, and the application must be notified with an error:

— Unit ID : the Unit ID of the confirmation does not match the Unit ID of the associated request ;

— Function code (service identifier): the Function code of the confirmation does not match the Function code of the associated request.

11 Application relationship protocol machines (ARPMs) for client/server

11.1 Application relationship protocol machines (ARPMs)

11.1.1 AR protocol machine (ARPM) for client AREP

11.1.1.1 Client ARPM states

The states of the Client ARPM and their descriptions are shown in Table 76 and Figure 28.

Table 76 – Client ARPM states

IDLE	The AREP is not active.
WAIT for CONFIRM	The AREP has sent a request PDU to a Server and is waiting for a confirmation from the Server.

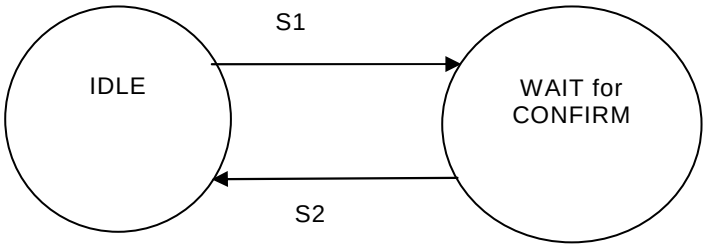

Figure 28 – State transition diagram of the Client ARPM

11.1.1.2 Client ARPM state table

Table 77 specifies the Client ARPM state machine.

Table 77 – Client ARPM state table

#	Current state	**Event** or **condition** **=> action**	Next state

S1	IDLE	Transaction.req
		=>
		DTC_req {
		called_address := remote_address,
		dlsdu := user_data
		}
			WAIT for CONFIRM

S2	WAIT for CONFIRM	DTC_cnf
		=>
		Transaction.cnf {
		remote_address := responding_address,
		user_data := dlsdu
		}
			IDLE

NOTE DTC is Data Transmission Confirmed.

11.1.2 AR protocol machine (ARPM) for server AREP

11.1.2.1 Server ARPM states

The states of the Server ARPM and their descriptions are shown in Table 78 and Figure 29.

Table 78 – Server ARPM states

IDLE	The AREP is not active.
WAIT for RESPONSE	The AREP has received a PDU from DLL, has sent the corresponding indication to the AL user and is waiting for a response from the AL user.

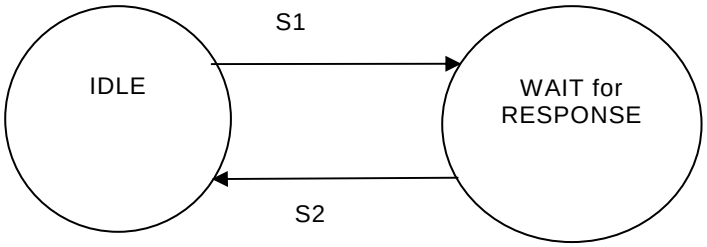

Figure 29 – State transition diagram of the server ARPM

11.1.2.2 Server ARPM state table

Table 79 specifies the Server ARPM state machine.

Table 79 – Server ARPM state table

#	Current state	**Event** or **condition** **=> action**	Next state

S1	IDLE	DTC_ind
		=>
		Transaction.ind {
		user_data := dlsdu
		}
			WAIT for RESPONSE

S2	WAIT for RESPONSE	Transaction.rsp
		=>
		DTC_rsp {
		dlsdu := user_data
		}
			IDLE

11.2 AREP state machine primitive definitions

11.2.1 Primitives exchanged between DMPM and ARPM

The primitives exchanged between the DMPM and the ARPM are specified in Table 80 and Table 81.

Table 80 – Primitives issued from ARPM to DMPM

Primitive names	Source	Associated parameters	Functions
DTC_req	ARPM	called_address, dlsdu	This primitive is used to request the DMPM to transfer an ALPDU to a Server device.
DTC_rsp	ARPM	dlsdu	This primitive is used to request the DMPM to transfer an ALPDU to a Client device.

Table 81 – Primitives issued by DMPM to ARPM

Primitive names	Source	Associated parameters	Functions
DTC_ind	DMPM	dlsdu	This primitive is used to pass an ALPDU received as a Data Like Layer service data unit to the Server ARPM.
DTC_cnf	DMPM	responding_address, dlsdu	This primitive is used to convey an ALPDU received from the Server device.

11.2.2 Parameters of ARPM/DMPM primitives

The parameters used with the primitives exchanged between the ARPM and the DMPM are described in Table 82.

Table 82 – Parameters used with primitives exchanged between ARPM and DMPM

Parameter name	Description
called_address	This parameter conveys the value of the address parameter supplied with the Data Like Layer primitive.
responding_address	This parameter conveys the value of the address parameter of the received primitive.
dlsdu	This parameter conveys the value of the data to be conveyed or received by data-link layer.

11.3 AREP state machine functions

ARPM does not use any functions.

12 DLL mapping protocol machine (DMPM) for client/server

12.1 AREP mapping to data link layer

12.1.1 General

This section describes the mapping of the AL to the Fieldbus data-link layer. It does not redefine the DLSAP attributes or DLME attributes that are defined in the data-link layer specification; rather, it defines how they are used by each of the AR classes. A means to configure and monitor the values of these attributes is provided by Network Management.
The following class definitions describe the DLSAP attributes and the DLME attributes required to support each of the AREP classes.
NOTE Undefined attributes use the same definitions as those previously defined.

12.1.2 DLL mapping of client AREP class

12.1.2.1 Client AREP class formal model

The DLL Mapping attributes and their permitted values and the DLL services used with the Client AREP class are defined in this subclause.

```
CLASS: Client
PARENT CLASS: Top
ATTRIBUTES:
1.        (m)  Attribute:     RemoteAddress
DLL SERVICES:
1.        (m)  OpsService:    Transmit_request
2.        (m)  OpsService:    Receive_confirm
```

12.1.2.2 Attributes

RemoteAddress
This attribute specifies the remote address to which request APDUs are sent, or from which confirm APDUs are received.

12.1.2.3 DLL services

12.1.2.3.1 Transmit_request

This service is used in the Client device to transfer an APDU to the remote Server DLL user.

12.1.2.3.2 Receive_confirm

The DLL uses this service to receive the response APDU from a remote Server DLL user.

12.1.3 DLL mapping of server AREP class

12.1.3.1 Server AREP class formal model

The DLL Mapping attributes and their permitted values and the DLL services used with the Server AREP class are defined in this subclause.

```
CLASS: Server
PARENT CLASS: Top
ATTRIBUTES:
```

1. (m) Attribute: Address
DLL SERVICES:
1. (m) OpsService: Receive_indication
2. (m) OpsService: Transmit_response

12.1.3.2 Attributes

Address
This attribute specifies the address that the Slave device uses to receive and send DLPDU.

12.1.3.3 DLL services

12.1.3.3.1 Receive_indication

The DLL uses this service to transfer the indication APDU from a remote Client DLL user.

12.1.3.3.2 Transmit_response

This service is used in the Server device to transfer an APDU to the remote Client DLL user.

12.2 DMPM states

The defined states and their descriptions of the DMPM are shown in Table 83 and Figure 30.

Table 83 – DMPM state descriptions

State Name	Description
ACTIVE	The DMPM in the ACTIVE state is ready to transmit or receive primitives to or from the Data Like Layer and the ARPM.

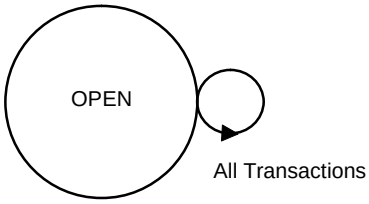

Figure 30 – State transition diagram of DMPM

12.3 DMPM state machine

Table 84 and Table 85 specify the DMPM state machine.

Table 84 – DMPM state table – client transactions

#	Current state	Event or condition => action	Next state

S1	ACTIVE	DTC_req
		=>
		Transmit.request {
		address := called_address,
		dl_dls_user_data := dlsdu
		}
			ACTIVE

S1	ACTIVE	Receive. confirm
		=>
		DTC_cnf {
		responding_address := address,
		dlsdu:= dl_dls_user_data
		}
			ACTIVE

Table 85 – DMPM state table – server transactions

#	Current state	Event or condition => action	Next state

S1	ACTIVE	DTC_rsp
		=>
		Transmit. response {
		dl_dls_user_data := dlsdu
		}
			ACTIVE

S1	ACTIVE	Receive. indication
		=>
		DTC_ind {
		dlsdu:= dl_dls_user_data
		}
			ACTIVE

12.4 Primitives exchanged between data link layer and DMPM

The primitives exchanged between the data-link layer and the DMPM are specified in Table 86.

Table 86 – Primitives exchanged between data-link layer and DMPM

Primitive names	Source	Associated parameters
Receive. indication	data-link layer	dl_dls_user_data
Receive. confirm	data-link layer	address, dl_dls_user_data
Transmit.request	DMPM	address, dl_dls_user_data
Transmit. response	DMPM	dl_dls_user_data

12.4.1 Functions used by DMPM

DMPM does not use any functions.

12.5 Client/server on TCP/IP

12.5.1 General

This section describes the client/server encapsulation when the DLL is TCP/IP.

12.5.2 TCP/IP encapsulation

The APDU for client/server is as described in this document in Clause 5.2, and illustrated in Figure 31.

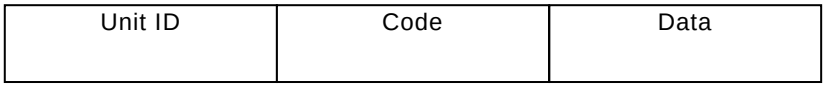

Figure 31 – APDU Format

TCP/IP encapsulation is obtained by adding a header to the APDU. The parameters of the header are described in Table 87.

Table 87 – Encapsulation parameters for client/server on TCP/IP

Parameter	Length	Description
Transaction Identifier	2 octets	Invoke_ID, allows pairing of request/response
Protocol Identifier	2 octets	0 for client/server
Length	2 octets	number of octets in APDU

The PDU carried as payload by TCP/IP becomes the one described in Figure 32.

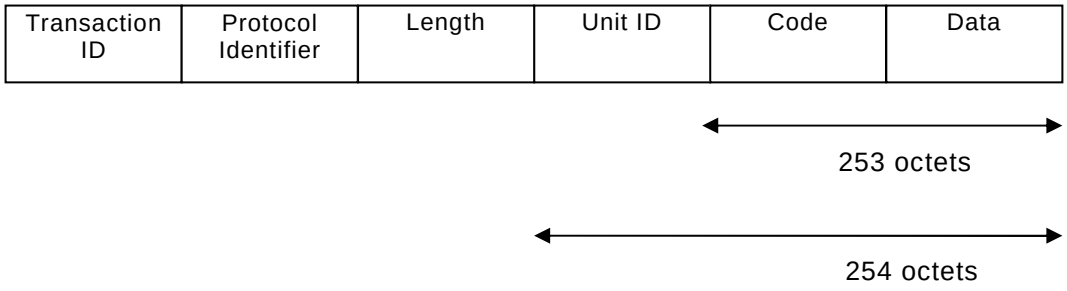

Figure 32 – TCP/IP PDU Format

The transaction identifier is the same as the Invoke ID, and it must be unique across all the identifiers still pending on the connection involved.

12.5.3 Assigned Port

client/server communicates using port 502, assigned by IANA.

12.5.4 Protocol Identifier

The protocol identifier for Type 15 client/server must be 0. The server must discard any transaction with a different protocol identifier.

12.5.5 Unit ID

On TCP/IP, when no gateways or IP co-located application entities are involved, the client and server are the intended end-points of the connection, and they are fully identified using the IP address. In this case the Unit ID may be ignored by the server, and the client should set it to the value of 255.

In case of gateways or IP co-located application entities, the Unit ID is used to identify the server connected to the gateway, or the server amongst the IP co-located application entities. In this case the value of 255 is recommended for addressing the gateway itself, or the IP device hosting the application entities.

12.5.6 TCP as a streaming protocol

The length field in the TCP envelope is used to identify the transaction payload boundaries, since TCP is a streaming protocol.

The server must be able to handle situations with several outstanding indications in pipelined transaction on the same connection, up to an implementation dependent number, usually

dictated by resource constraints. If such a number is exceeded the server must respond with Exception code = 0x06: Server Busy.

The above limit may be per connection, or it may be a shared limit across the connections on the same server.

The streaming nature of the TCP protocol allows for cases where the server received only a partial transaction, according to a valid length. The server must be able to buffer the partial transaction and wait for the remaining payload. The server may implement mechanisms, e.g. via a timer, to reclaim resources if the wait exceed a configured time.

12.5.7 [Informative] TCP interfaces and parameterization

The Berkeley Software Distribution (BSD) Socket Interface is often used to communicate using TCP (see for example “TCP/IP Illustrated, Volume 2, Gary R. Wright and W. Richard Stevens”).

A socket is an endpoint of communication. After the establishment of the TCP connection the data can be transferred. The **send()** and **recv()** functions are designed specifically to be used with sockets that are already connected.

The **setsockopt ()** function allows a socket’s creator to associate options with a socket. These options modify the behavior of the socket. The description of these options and recommended settings useful for Type 15 client/server will follow.

12.5.7.1 Connection parameters

SO-RCVBUF, **SO-SNDBUF**:

These parameters allow setting the high water mark for the send and the receive sockets. They can be adjusted for flow control management. The size of the receive buffer is the maximum size advertised window for that connection. Socket buffer sizes must be increased in order to increase performances. Nevertheless these values must be smaller than internal driver resources in order to close the TCP window before exhausting internal driver resources.

The receive buffer size depends on the TCP Windows size, the TCP Maximum segment size and the time needed to absorb the incoming frames. With a Maximum Segment Size of 300 octets (easily accommodating a Type 15 client/server request), to accommodate 3 frames, the socket buffer size value can be adjusted to 900 octets.

TCP-NODELAY:

Small packets (called tinygrams) are normally not a problem on LANs, since most LANs are not congested, but these tinygrams can lead to congestion on wide area networks. A simple solution, called the "NAGLE algorithm", is to collect small amounts of data and sends them in a single segment when the TCP acknowledgments of a previous packet arrive.

In order to have better behavior it is recommended to send small amounts of data directly without trying to gather them in a single segment. That is why it is recommended to force the TCP-NODELAY option that disables the "NAGLE algorithm" on client and server connections.

SO-REUSEADDR:

When a Type 15 server closes a TCP connection initialized by a remote client, the local port number used for this connection cannot be reused for a new opening while that connection stays in the "Time-wait" state (during two MSL : Maximum Segment Lifetime).

It is recommended to specify the SO-REUSEADDR option for each client and server connection to bypass this restriction. This option allows the process to assign itself a port number that is part of a connection that is in the 2MSL wait for client and listening socket.

SO-KEEPALIVE:

By default on the TCP/IP protocol there is no data sent across an idle TCP connection. Therefore if no process at the ends of a TCP connection is sending data to the other, nothing is exchanged.

Under the assumption that either the client application or the server application uses timers to detect inactivity in order to close a connection, it is recommended to enable the KEEPALIVE option on both client and server connections in order to poll the other end to know its status.

Nevertheless it must be considered that enabling KEEPALIVE can cause perfectly good connections to be dropped during transient failures, and that it consumes unnecessary bandwidth if the keep alive timer is too short.

12.5.7.2 TCP layer parameters

Time Out on establishing a TCP Connection:

Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection, this default value should be adapted to the constraint of the application.

Keep alive parameters:

The default idle time for a connection is 2 hours. Idles times in excess of this value trigger a keep alive probe. After the first keep alive probe, a probe is sent every 75 seconds for a maximum number of times unless a probe response is received.

The maximum number of keep alive probes sent out on an idle connection is 8. If no probe response is received after sending out the maximum number of keep alive probes, TCP signals an error to the application that can decide to close the connection.

Time-out and retransmission parameters:

A TCP packet is retransmitted if its loss has been detected. One way to detect the loss is to manage a Retransmission Time-Out (RTO) that expires if no acknowledgement has been received from the remote side.

TCP manages a dynamic estimation of the RTO. For that purpose a Round-Trip Time (RTT) is measured after the sending of every packet that is not a retransmission. The Round-Trip Time (RTT) is the time taken for a packet to reach the remote device and to get back an acknowledgement to the sending device. The RTT of a connection is calculated dynamically, nevertheless if TCP cannot get an estimate within 3 seconds, the default value of the RTT is set to 3 seconds.

If the RTO has been estimated, it applies to the sending of the next packet. If the acknowledgement of the next packet is not received before the estimated RTO expiration, the ‘Exponential BackOff’ (detailed below) is activated. A maximum number of retransmissions of the same packet are allowed during a certain amount of time. After that if no acknowledgement has been received, the connection is aborted.

Some TCP/IP stacks allow the set-up of the maximum number of retransmissions and the maximum amount of time before the abort of the connection.

Some retransmission algorithms are defined in TCP IETF RFCs and other papers:

— The **Jacobson's RTO estimation algorithm** is used to estimate the Retransmission Time-Out (RTO);

— The **Karn's algorithm** says that the RTO estimation should not be done on a retransmitted segment;

— The **Exponential BackOff** defines that the retransmission time-out is doubled for each retransmission with an upper limit of 64 seconds;

— The **fast retransmission algorithm** allows retransmitting after the reception of three duplicate acknowledgments. This algorithm is advised because on a LAN it may lead to a quicker detection of the loss of a packet than waiting for the RTO expiration.

The use of these algorithms is recommended for a Type 15 client/server implementation. They are described in “TCP/IP Illustrated, Volume 2, Gary R. Wright and W. Richard Stevens”, which also points to the original sources.

13 AP-Context state machines for publish/subscribe

There is no AP-Context State Machine in this Type of FAL.

14 Protocol machines for publish/subscribe

14.1 General

publish/subscribe communicates using the following characteristics:

— the transport has a generalized notion of a unicast address (shall fit within 16 octets);

— the transport has a generalized notion of a port (shall fit within 4 octets), e.g. could be a UDP port, an offset in a shared memory segment, etc.;

— the transport can send a datagram (uninterpreted sequence of octets) to a specific address/port;

— the transport can receive a datagram at a specific address/port;

— the transport will drop messages if incomplete or corrupted during transfer (i.e. publish/subscribe assumes messages are complete and not corrupted);

— the transport provides a means to deduce the size of the received message.

Publish/subscribe exchanges messages composed of service requests as detailed in 7.2.

The Figure 33 illustrates the receiver state machine, according to the APDU interpretation described in 7.4.

There is only one APDU, composed of al the requests, and it is sent as an unconfirmed service.

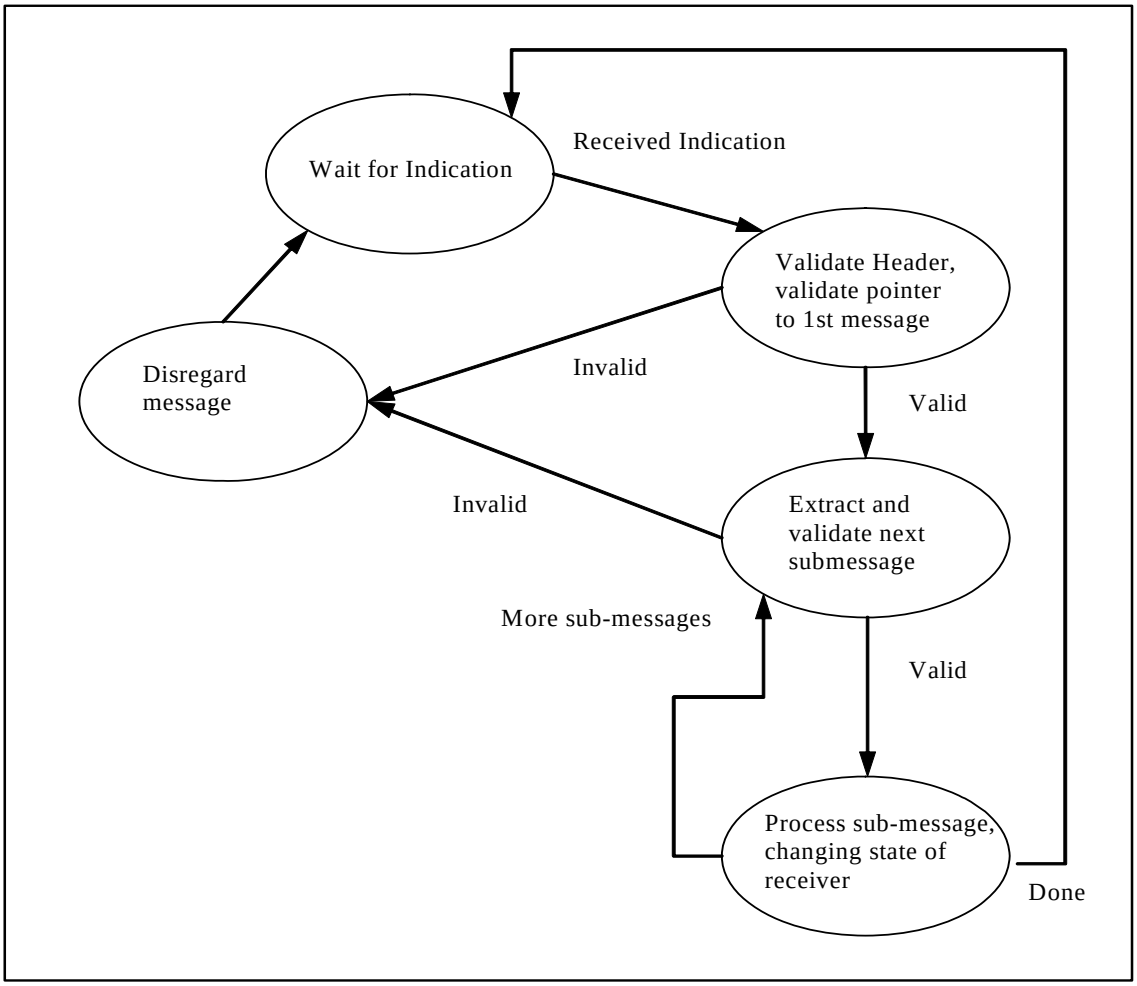

Figure 33 – Publish/subscribe receiver

The general extraction and validation process for a sub-message is according to 7.4.2 and it is repeated here for convenience:

a) The last two octets of a sub-message header, the *octetsToNextHeader* field, contains the number of octets to the next sub-message. If this field is invalid, the rest of the message is invalid;

b) The first octet of a sub-message header is the *sub-messageID*. A sub-message with an unknown ID must be ignored and parsing must continue with the next sub-message. Concretely: an implementation of publish/subscribe 1.0 must ignore any sub-messages with IDs that are outside of the sub-messageID list used by version 1.0. IDs in the vendor-specific range coming from a *vendorID* that is unknown must be ignored and parsing must continue with the next sub-message;

c) The second octet of a sub-message header contains flags; unknown flags should be skipped. An implementation of publish/subscribe 1.0 should skip all flags that are marked as 'X' (unused) in the protocol;

d) A valid *octetsToNextHeader* field must *always* be used to find the next sub-message, even for sub-messages with unknown IDs;

In addition to the general rules, there are sub-message specific rules. A known but invalid sub-message invalidates the rest of the message. 7.5.1 through 7.5.10.1 each describe a known sub-message and when it should be considered invalid.

The processing of a sub-message has the following effects:

— It can change the state of the receiver; this state influences how the following sub-messages in the message are interpreted. 7.5.1 through 7.5.10.1 show how the state changes for each sub-message. In this version of the protocol, only the Header and the sub-messages INFO_SRC, INFO_REPLY and INFO_TS change the state of the receiver;

— The sub-message, interpreted within the message, has a logical interpretation: it encodes one of the five basic publish/subscribe services: ACK, GAP, HEARTBEAT, ISSUE or VAR, beside the logistic services. The logical interpretation is also described in 7.5.1 through 7.5.10.1.

14.2 Publish/subscribe on UDP

14.2.1 General

14.2.1.1 publish/subscribe and the UDP payload

When publish/subscribe is used over UDP/IP, a message is the contents (payload) of exactly one UDP/IP Datagram.

14.2.1.2 UDP/IP Destinations

A UDP/IP destination consists of an IPAddress and a Port. This document uses notation such as "12.44.123.92:1024" or "225.0.1.2:6701" to refer to such a destination. The IP address can be a unicast or multicast address.

14.2.1.3 Note on relative addresses

The publish/subscribe protocol often sends IP addresses to a sender of messages, so that the sender knows where to send future messages. These destinations are always interpreted locally by the sender of UDP datagrams. Certain IP addresses, such as "127.0.0.1" have only relative meaning (i.e. they do not refer to a unique host).

14.2.2 Well known ports

At the network level, publish/subscribe uses the following three well-known ports:

wellknownManagerPort = portBaseNumber + 10 * portGroupNumber

wellknownUsertrafficMulticastPort = 1 + portBaseNumber + 10 * portGroupNumber

wellknownMetatrafficMulticastPort = 2 + portBaseNumber + 10 * portGroupNumber

Within a network, all applications need to use the same *portBaseNumber*. Applications that want to communicate with each other use the same *portGroupNumber*; applications that need to be isolated from each other use a different *portGroupNumber*.

Each application needs to be configured with the correct *portBaseNumber* and *portGroupNumber*.

Except for the rules stated above, publish/subscribe does not define which *portBaseNumber* and *portGroupNumber* are used nor how the applications participating in a network obtain this information, but the base number 7400 should be the one used, since the ports 7400, 7401 and 7402 are the one assigned by IANA for publish/subscribe usage.

Bibliography

IEC/TR 61158-1:2010[2], *Industrial communication networks – Fieldbus specifications – Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series*

ISO/IEC 7498-3, *Information technology – Open Systems Interconnection – Basic Reference Model: Naming and addressing*

OMG UML 2.0 Superstructure Specification, available at <http://www.omg.org>

GARY R. WRIGHT and W. RICHARD STEVENS, *TCP/IP Illustrated, Volume 2,*

Modbus-IDA: Modbus Messaging on TCP/IP Implementation Guide, available at <http://www.Modbus-IDA.org>

_____

[2] To be published.

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

3, rue de Varembé
PO Box 131
CH-1211 Geneva 20
Switzerland

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