IEC 61158-6-2

Edition 1.0 2007-12

INTERNATIONAL STANDARD

Industrial communication networks – Fieldbus specifications – Part 6-2: Application layer protocol specification – Type 2 elements

IEC 61158-6-2:2007(E)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

About the IEC

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

About IEC publications

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

- 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 It also gives information on projects, withdrawn and replaced publications.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 61158-6-2

Edition 1.0 2007-12

INTERNATIONAL STANDARD

Industrial communication networks – Fieldbus specifications – Part 6-2: Application layer protocol specification – Type 2 elements

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PRICE CODE XH

ICS 35.100.70; 25.040.40

ISBN 2-8318-9474-3

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

INDUSTRIAL COMMUNICATION NETWORKS – FIELDBUS SPECIFICATIONS –

Part 6-2: Application layer protocol specification – Type 2 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 provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
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 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 IEC 61784 series. Use of the various protocol types in other combinations may require permission from their respective intellectual-property-right holders.

International Standard IEC 61158-6-2 has been prepared by subcommittee 65C: Industrial networks, of IEC technical committee 65: Industrial-process measurement, control and automation.

This first edition and its companion parts of the IEC 61158-6 subseries cancel and replace IEC 61158-6:2003. This edition of this part constitutes a technical revision. This part and its Type 2 companion parts also cancel and replace IEC/PAS 62413, published in 2005

This edition of IEC 61158-6 includes the following significant changes from the previous edition:

a) deletion of the former Type 6 fieldbus for lack of market relevance;
b) addition of new types of fieldbuses;

c) partition of part 6 of the third edition into multiple parts numbered -6-2, -6-3, …

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

FDIS	Report on voting
65C/476/FDIS	65C/487/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.

The committee has decided that the contents of this publication will remain unchanged until the maintenance result 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 The revision of this standard will be synchronized with the other parts of the IEC 61158 series.

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

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 implementors 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 6-2: Application layer protocol specification – Type 2 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 2 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 specifies interactions between remote applications and defines the externally visible behavior provided by the Type 2 fieldbus application layer in terms of

a) the formal abstract syntax defining the application layer protocol data units conveyed between communicating application entities;
b) the transfer syntax defining encoding rules that are applied to the application layer protocol data units;
c) the application context state machine defining the application service behavior visible between communicating application entities;
d) the application relationship state machines defining the communication behavior visible between communicating application entities.

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

1) define the wire-representation of the service primitives defined in IEC 61158-5-2, and
2) define the externally visible behavior associated with their transfer.

This standard specify the protocol of the Type 2 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 61158-5-2.

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 61158-3-2, *Industrial communication networks – Fieldbus specifications – Part 3-2: Data-link layer service definition – Type 2 elements*

IEC 61158-4-2, *Industrial communication networks – Fieldbus specifications – Part 4-2: Data-link layer protocol specification – Type 2 elements*

IEC 61158-5-2, *Industrial communication networks – Fieldbus specifications – Part 5-2: Application layer service definition – Type 2 elements*

IEC 61588:2004[1], *Precision clock synchronization protocol for networked measurement and control systems*

IEC 61784-3-2, *Industrial communication networks – Profiles – Functional safety fieldbuses – Part 3-2: Additional specifications for CPF 2*

IEC 62026-3[2], *Low-voltage switchgear and controlgear – Controller-device interfaces (CDIs) – Part 3: DeviceNet*

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

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

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

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

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

ISO/IEC 10731, *Information technology – Open Systems Interconnection – Basic Reference Model – Conventions for the definition of OSI services*

ISO 11898:1993[3], *Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication*

[1] Compliance with future editions of this standard will need checking.

[2] To be published.

[3] A newer edition of this standard has been published, but only the cited edition applies.

3 Terms, definitions, symbols, abbreviations and conventions

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

3.1 Terms and definitions from other ISO/IEC standards

3.1.1 Terms and definitions from ISO/IEC 7498-1

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

3.1.2 Terms and definitions from ISO/IEC 9545

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

3.1.3 Terms and definitions from ISO/IEC 8824

a) object identifier
b) type
c) value
d) simple type
e) structured type
f) component type
g) tag
h) Boolean type
i) true
j) false
k) integer type
l) bitstring type
m) octetstring type
n) null type
o) sequence type
p) sequence of type
q) choice type
r) tagged type
s) any type
t) module
u) production

3.1.4 Terms and definitions from ISO/IEC 8825

a) encoding (of a data value)
b) data value
c) identifier octets (the singular form is used in this standard)
d) length octet(s) (both singular and plural forms are used in this standard)
e) contents octets

3.2 Terms and definitions from IEC 61158-5-2

a) application relationship
b) client
c) peer
o) server

3.3 Additional terms and definitions

3.3.1
allocate
take a resource from a common area and assign that resource for the exclusive use of a specific entity

3.3.2
application
function or data structure for which data is consumed or produced

3.3.3
application objects
multiple object classes that manage and provide a run time exchange of messages across the network and within the network device

3.3.4
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 behaviour of an object. Attributes are divided into class attributes and instance attributes.

3.3.5
behaviour
indication of how an object responds to particular events

3.3.6
boundary clock
clock with more than a single PTP port, with each PTP port providing access to a separate PTP communication path

NOTE Boundary clocks are used to eliminate fluctuations produced by routers and similar network elements.

3.3.7
called
service user or a service provider that receives an indication primitive or a request APDU

3.3.8
calling
service user or a service provider that initiates a request primitive or a request APDU

3.3.9
class
set of objects, all of which represent the same kind of system component

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

3.3.10
class attributes
attribute that is shared by all objects within the same class

3.3.11
class code
unique identifier assigned to each object class

3.3.12
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

3.3.13
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

3.3.14
clock
device providing a measurement of the passage of time since a defined epoch

NOTE There are two types of clocks in IEC 61588:2004, boundary clocks and ordinary clocks.

3.3.15
communication objects
components that manage and provide a run time exchange of messages across the network

EXAMPLES Connection Manager object, Unconnected Message Manager (UCMM) object, Message Router object.

3.3.16
connection
logical binding between application objects that may be within the same or different devices

NOTE Connections may be either point-to-point or multipoint.

3.3.17
connection ID (CID)
identifier assigned to a transmission that is associated with a particular connection between producers and consumers, providing a name for a specific piece of application information

3.3.18
connection path
an octet stream which defines the application object to which a connection instance applies

3.3.19
connection point
buffer which is represented as a subinstance of an Assembly object

3.3.20
consume
act of receiving data from a producer

3.3.21
consumer
node or sink that is receiving data from a producer

3.3.22
consuming application
application that consumes data

3.3.23
cyclic
repetitive in a regular manner

3.3.24
device
physical hardware connected to the link

NOTE A device may contain more than one node.

3.3.25
device profile
collection of device dependent information and functionality providing consistency between similar devices of the same device type

3.3.26
end node
producing or consuming node

3.3.27
end point
one of the communicating entities involved in a connection

3.3.28
epoch
reference time defining the origin of a time scale
[IEC 61588:2004]

3.3.29
error
discrepancy between a computed, observed or measured value or condition and the specified or theoretically correct value or condition

3.3.30
frame
denigrated synonym for DLPDU

3.3.31
grandmaster clock
clock which serve as the primary source of time to which all others (within a collection of IEC 61588 clocks) are ultimately synchronized.

3.3.32
instance
the actual physical occurrence of an object within a class, identifying one of many objects within the same object class.

EXAMPLE California is an instance of the object class state.

NOTE The terms object, instance, and object instance are used to refer to a specific instance.

3.3.33
instance attribute
attribute that is unique to an object instance and not shared by the object class

3.3.34
instantiated
object that has been created in a device

3.3.35
interoperability
capability of User Layer entities to perform coordinated and cooperative operations using the services of the FAL

3.3.36
Keeper
object responsible for distributing link configuration data to all nodes on the link

3.3.37
little endian
Describes a model of memory organisation which stores the least significant octet at the lowest address, or for transfer, which transfers the lowest order octet first

3.3.38
Lpacket
a piece of application information that contains a size, control octet, tag, and link data.

NOTE Peer data-link layers use Lpackets to send and receive service data units from higher layers in the OSI stack.

3.3.39
management information
network accessible information that supports the management of the Fieldbus environment

3.3.40
master clock
single clock which serves as the primary source of time within each region, and which will in turn synchronize to other master clocks and ultimately to the grandmaster clock

NOTE A system of IEC 61588 clocks may be segmented into regions separated by boundary clocks.

3.3.41
member
piece of an attribute that is structured as an element of an array

3.3.42
Message Router
object within a node that distributes messaging requests to appropriate application objects

3.3.43
multipoint connection
connection from one node to many

NOTE Multipoint connections allow messages from a single producer to be received by many consumer nodes.

3.3.44
network
a set of nodes connected by some type of communication medium, including any intervening repeaters, bridges, routers and lower-layer gateways

3.3.45
object
abstract representation of a particular component within a device, usually a collection of related data (in the form of variables) and methods (procedures) for operating on that data that have clearly defined interface and behaviour

3.3.46
object specific service
service unique to the object class which defines it

3.3.47
ordinary clock
IEC 61588 clock with a single PTP port

3.3.48
originator
client responsible for establishing a connection path to the target

3.3.49
point-to-point connection
connection that exists between exactly two application objects

3.3.50
Precision Time Protocol (PTP)
name used for the time synchronization protocol.

3.3.51
produce
act of sending data to be received by a consumer

3.3.52
producer
node that is responsible for sending data

3.3.53
PTP message
IEC 61588 time synchronization message

NOTE There are five designated messages types defined by IEC 61588:2004: Sync, Delay_Req, Follow-up, Delay_Resp, and Management.

3.3.54
PTP port
logical access point for IEC 61588 communications to the clock containing the port

3.3.55
receiving
service user that receives a confirmed primitive or an unconfirmed primitive, or a service provider that receives a confirmed APDU or an unconfirmed APDU

3.3.56
resource
resource is a processing or information capability of a subsystem

3.3.57
sending
service user that sends a confirmed primitive or an unconfirmed primitive, or a service provider that sends a confirmed APDU or an unconfirmed APDU

3.3.58
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

3.3.59
service
operation or function than an object and/or object class performs upon request from another object and/or object class

3.3.60
synchronized clocks
(to a specified uncertainty) two clocks which have the same epoch and for which measurements of any time interval by both clocks differ by no more than the specified uncertainty

NOTE The timestamps generated by two synchronized clocks for the same event will differ by no more than the specified uncertainty.

3.3.61
System Time
absolute time value as defined by CPF2 time synchronization in the context of a distributed time system where all devices have a local clock that is synchronized with a common master clock

NOTE In the context of CPF2, System Time is a 64-bit integer value in units of nanoseconds with a value of 0 corresponding to the date 1970-01-01.

3.3.62
target
end-node to which a connection is established

3.3.63
temporary node
transient node

3.3.64
transaction id
field within a UCMM header that matches a response with the associated request

3.3.65
Unconnected Message Manager (UCMM)
component within a node that transmits and receives unconnected explicit messages and sends them directly to the Message Router object

3.3.66
unconnected service
messaging service which does not rely on the set up of a connection between devices before allowing information exchanges

3.4 Abbreviations and symbols

ASCII	American Standard Code for Information Interchange
CID	connection ID
DLL	data-link layer
PDU	protocol data unit
PTP	Precision Time Protocol [IEC 61588:2004]
OSI	open systems interconnection (see ISO/IEC 7498 series)
Rcv	receive
Rx	receive
SDU	service data unit
SEM	state event matrix
STD	state transition diagram, used to describe object behaviour
TPDU	transport protocol data unit
Tx	transmit
Xmit	transmit
CM_API	actual packet interval
O2T or O⇒T	originator to target (connection parameters)
CM_RPI	requested packet interval
TUI	table unique identifier
T2O or T⇒O	target to originator (connection parameters)

3.5 Conventions

3.5.1 General concept

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 61158-5-2. 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 61158-5-2. The service specification defines the services that are provided by the ASE.

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

3.5.1.1 Attribute specification

Attributes are defined in an Attribute Table using the format and terms defined in Figure 1.

Attribute ID	Name	Data type	Semantics of values

Figure 1 – Attribute table format and terms

The **Attribute ID** shall be a unique integer identification value assigned to an attribute. The valid ranges for Attribute IDs shall be as specified in 5.1.1.10.1.3 The Attribute ID shall identify the particular attribute being accessed.

Name shall refer to the name assigned to the attribute. An attribute may contain sub-elements, which may also have names, as is the case with the **STRUCT of** data type. The attribute name shall be the name which appears first, or at the top row in the name column (the row that contains the Attribute ID).

Every attribute shall be assigned a **Data type** which shall be either an elementary or a derived data type. The data types specified for the defined attributes shall be used in all implementations.

NOTE The elementary data types are defined in IEC 61158-5-2.

Semantics of values shall specify the meaning of the value(s) of the attribute. If this information needs more room than can fit in the table it will immediately follow the Class Attribute table. Included in the Class Attribute table will be an appropriate reference to this information.

If a Class Attribute is *optional*, then a default value or a special case processing method shall be defined such that the Client (Requester) can process the error message that occurs when accessing those objects that choose not to implement the class attribute.

3.5.1.2 Common services

3.5.1.2.1 Service_PDU definitions

Each service has unique parameters for request and response. The service request and response parameters shall be defined using service Request/Response parameter tables as defined in Figure 2.

Name	Type	Semantics of values

Figure 2 – Service request/response parameter

Name shall refer to the name given to the service request/response parameter.

Type shall specify the data type of the service request/response parameter.

Semantics of values shall specify the meaning of the values of the service request/response parameter, e.g. “the value is counts of microseconds.”

3.5.1.2.2 Get_Attribute_All response

3.5.1.2.2.1 General definition

When the Get_Attribute_All common service is included in the list of supported System/Object Management services for an object class, then the **Get_Attribute_All** response shall be detailed for this class : the sequence or order of the data returned in the Service_ResponsePDU shall be specified. There are three ways in which the Get_Attribute_All Service_ResponsePDU may be specified:

- list the ordering of the attributes in the response message;
- specify the actual data array of the response message;
- combine the above two methods such that the actual data array also contains the attribute numbers.

Whichever method is used, the rules specified in Table 1 shall be adhered to when specifying the Get_Attribute_All Service_ResponsePDU of an object class for both the Class Attributes and the Instance Attributes.

Table 1 – Get_Attribute_All response service rules

Rule number	Rule
1	If the definition of the Get_Attribute_All response includes optional attributes, then default values shall be specified in the response description. These defaults shall be used if an implementation does not support the optional attributes. If any of the following optional attributes are included in an object specification, but not supported in the implementation of the object, then the following shall be adhered to: – if the class attribute "Optional attribute list" is not supported, the default value of zero shall be inserted into the response array and no optional attribute numbers shall follow; – if the class attribute "optional service list" is not supported, the default value of zero shall be inserted into the response array and no optional service numbers shall follow.
2	If new attributes are added to an existing object, those attributes shall be added to the end of the response attribute list or data array to ensure compatibility with different object revisions.
3	Whichever method is used to specify the response, it shall be done in such a way as to be unambiguous, including rules to deal with variable length fields and padding.
4	The Get_Attribute_All response shall include only the open attributes; it shall not include any vendor specific attributes.

Table 2 is an example of the attribute ordering method of specifying the service data portion of a Get_Attribute_All response for class level attributes of an object which supports **optional gettable** class attributes 1,2,3 and 4.

Table 2 – Example class level object/service specific response data of Get_Attribute_All

Class attribute ID	Attribute name and default value
1	Revision, default = 0x0001
2	Max Instance, default = 0x0000
3	Number of Instances, default = 0x0000
4	Attribute list, number of attributes, default = 0x0000

Table 3 is an example of the data array method of specifying the service data portion of a Get_Attribute_All response for class level attributes of an object which supports **optional gettable** class attributes 1, 2, 3 and 4.

Table 3 – Example Get_Attribute_All data array method

Octet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Revision (low octet) Default = 1							
1	Revision (high octet) Default = 0							
2	Max Instance (low octet) Default = 0							
3	Max Instance (high octet) Default = 0							
4	Number of Instances (low octet) Default = 0							
5	Number of Instances (high octet) Default = 0							
6	Optional Attribute List : number of attributes (low octet) Default = 0							
7	Optional Attribute List : number of attributes (high octet) Default = 0							
8	Optional Attribute List : optional attribute #1 (low octet)							
9	Optional Attribute List : optional attribute #1 (high octet)							
2n + 6	Optional Attribute List : optional attribute #n (low octet)							
2n + 7	Optional Attribute List : optional attribute #n (high octet)							

3.5.1.2.2.2 Revisions

The defined Get_Attribute_All response for an object may increase in size with each revision of the object; however, to insure interoperability, the format of the first part of the response shall remain parseable by a client of the older revision of the object. Clients (Requesters) need not make use of this compatibility requirement.

NOTE The Revision class attribute is not the revision of an implementation (which is reflected in the Identity object, Minor/Major revision status bits), but the revision of the *class definition.*

3.5.1.2.3 Set_Attribute_All request

3.5.1.2.3.1 General definition

When the Set_Attribute_All common service is included in the list of supported common services, then the format of the **Set_Attribute_All request** shall be included in the object specification. The sequence or order of the data supplied in the Service Data portion of the request shall be specified. There are three ways in which the Set_Attribute_All request may be specified:

- list the order of the attributes in the request message;
- specify the actual data array of the request message;
- combine the above two methods such that the actual data array also contains the attribute numbers.

Whichever method is used, the rules specified in Table 4 shall be adhered to when specifying an object's Set_Attribute_All request in the object specification for both the Class Attributes and the Instance Attributes.

Table 4 – Set_Attribute_All request service rules

Rule number	Rule
1	An object shall support the Set_Attribute_All service only if all settable attributes shown in the Set_Attribute_All request are implemented as settable.
2	If new settable attributes are added to an existing object, those attributes shall be added to the end of the request attribute list or data array.
3	Whichever method is used to specify the response, it shall be done in such a way as to be unambiguous, including rules to deal with variable length fields and padding.
4	The Set_Attribute_All request shall include only the open attributes. It shall not include any vendor specific attributes.

Table 5 is an example of the attribute ordering method of specifying the service data portion of a Set_Attribute_All request for instance level attributes of an object which supports required settable instance attributes 7,8,9,10,11 and 12.

Table 5 – Example Set_Attribute_All attribute ordering method

Class Attribute ID	Attribute name
7	Output Range
8	Value Data Type
9	Fault State
10	Idle State
11	Fault Value
12	Idle Value

Table 6 is an example of the data array method of specifying the service data portion of a Set_Attribute_All request for instance level attributes of an object which supports required settable instance attributes 7, 8, 9, 10, 11 and 12.

Table 6 – Example Set_Attribute_All data array method

Octet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Output Range							
1	Value Data Type							
2	Fault State							
3	Idle State							
4	Fault Value (low octet)							
5	Fault Value (high octet)							
6	Idle Value (low octet)							
7	Idle Value (high octet)							

3.5.1.2.3.2 Revisions

A server processing a Set_Attribute_All request may need to process more data than is expected by the server if the client has implemented a newer revision of this object while the server an older revision. As a result, the server object may have to process a Set_Attribute_All request that contains more data because the additional attributes were not recognised. If the server receives more data in a Set_Attribute_All request than it expects the server shall respond with a general status code equal to *Too much data* (0x15).

NOTE The Revision class attribute is not the revision of an implementation (which is reflected in the Identity object, Minor/Major revision status bits), but the revision of the *class definition*.

3.6 Conventions used in state machines

3.6.1 State machine conventions for Type 2

3.6.1.1 General

State changes may be triggered by events (internal or external) or service invocations. Reaction to service invocations may depend on the value(s) of the attribute(s) accessed by the service.

Behaviour of the state machine shall be defined for combinations of :

- the *event* the machine receives;
- the *state* the machine is in when it receives notification of a state-changing event.

To define behaviour in these terms, a State Transition Diagram (STD) and a State Event Matrix (SEM) are used when applicable in the state machine specification.

3.6.1.2 State Transition Diagram (STD)

An ***event*** is an external stimulus that may cause a state transition. An STD graphically illustrates the states of an object and includes events, service calls and changes of attributes that cause it to transition to another state. Figure 3 shows an example of an STD.

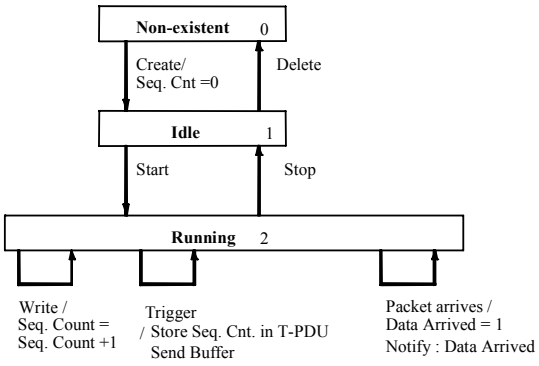


Figure 3 – Example of an STD

NOTE Event that is the primary cause of the State Transition is followed by "/" mark. Notification to upper layer is marked by "Notify :"

3.6.1.3 State Event Matrix

A ***state*** is the current active mode of operation of the state machine object (e.g. Running, Idle).

A state event matrix is a table that lists all possible events, services and changes in attribute values that initiate a state change, and indicates the response by the object to the event based on the state of the object when it receives notification of that event.

State event matrix format is as shown in Table 7.

Table 7 – State event matrix format

Event	State		
	State 1	....	State n
Event A description	Function triggered by event A in State 1 (if any) Notification to FAL user (if any) Transition to another state (if any)		Function triggered by event A in State n (if any) Notification to FAL user (if any) Transition to another state (if any)
.....	.....		.....
Event X description	Function triggered by event X in State 1 (if any) Notification to FAL user (if any) Transition to another state (if any)		Function triggered by event X in State n (if any) Notification to FAL user (if any) Transition to another state (if any)
In absence of function, notification or transition, the corresponding entry shall not be made in the table.			

3.6.1.4 Example state event matrix

Table 8 shows an example of a state machine with three states:

- *Non-existent*: the object has not yet been created; objects transition to the existent state via the create service (if the object may be dynamically created) or at power-up (if the object is fixed by design/implementation);
- *Idle:* the object accepts services (e.g. Get_Attribute_Single), but does not produce or consume data onto or from the link;
- *Running*: the object is performing all its specified functions.

Table 8 – Example state event matrix

Event	State		
	Non-existent	Idle	Running
Create	Transition to Idle	Error: Object already exists.	Error: Object already exists.
Delete	Error: Object does not exist. (General Error Code 0x16)	Transition to Non-existent	Error: Object State Conflict (General Error Code 0x0C)
Start	Error: Object does not exist. (General Error Code 0x16)	Transition to Running	Error: Object State Conflict (General Error Code 0x0C)
Stop	Error: Object does not exist. (General Error Code 0x16)	Error: Object State Conflict (General Error Code 0x0C)	Transition to Idle
Get_Attribute_Single	Error: Object does not exist. (General Error Code 0x16)	Validate/service the request. Return response	Validate/service the request. Return response
Set_Attribute_Single	Error: Object does not exist. (General Error Code 0x16)	Validate/service the request. Return response	Error: Object State Conflict (General Error Code 0x0C)

4 Abstract syntax

4.1 FAL PDU abstract syntax

4.1.1 General

FAL_PDU shall be either a UCMM_PDU or a Transport_PDU.

UCMM_PDU is used to convey information for connection-less services.

Transport_PDU is used to convey information for connection-oriented services.

A connected message assumes previously negotiated resources and parameters at its source, its destination(s), and any intermediate transit points. These resources are referenced by a unique connection identifier and do not need to be contained in each message, only the connection ID is needed to identify the message and refer to its related parameters, thus giving significant savings in message efficiency.

An unconnected message provides a means to communicate on the local link without previously negotiated resources at the destination so it shall carry full destination ID details, internal data descriptors and full source ID details if a reply is requested. Unconnected messages are used mainly to create connections.

The unconnected service is provided by the Unconnected Message Manager (UCMM). Messages received through the UCMM are forwarded to the Message Router (MR), which direct them to the appropriate internal object for execution. Connections are established by specific unconnected messages sent through the Connection Manager (CM) using UCMM services. Connections may be established either to the Message Router (for messaging purpose), or directly to an application object. Connection target is specified using a connection path, and may be either on the local or a remote link, through several intermediate router nodes. Once a connection is established with an application object, the UCMM, MR and CM are no longer required, since data will be exchanged directly with the connected objet, based on the corresponding connection ID. Connected messages sent to the Message Router will be forwarded to the appropriate internal object for execution.

4.1.2 PDU structure

4.1.2.1 UCMM_PDU structure

UCMM_PDU is a sequence of a UCMM_Header, followed by either OM_Service (Object Management ASE) or CM_Service (Connection Manager ASE).

OM_Service shall be either OM_Request or OM_Response.

OM_Request shall consist of MR_Request_Header and a Service_RequestPDU.

Service_RequestPDU shall be one of the following :

- Get_Attribute_Single_RequestPDU
- Get_Attribute_All_RequestPDU
- Get_Attribute_List_RequestPDU
- Set_Attribute_Single_RequestPDU
- Set_Attribute_All_RequestPDU
- Set_Attribute_List_RequestPDU
- Reset_RequestPDU
- Create_RequestPDU
- Delete_RequestPDU
- Start_RequestPDU
- Stop_RequestPDU
- Find_Next_Object_Instance_RequestPDU
- NOP_RequestPDU
- Apply_Attributes_RequestPDU
- Save_RequestPDU
- Restore_RequestPDU
- Group_Sync_RequestPDU
- Add_AckData_Path_RequestPDU
- Remove_AckData_Path_RequestPDU
- Initialize_RequestPDU
- Management_Message_RequestPDU
- Connection_Bind_RequestPDU
- Producing_Application_Lookup_RequestPDU

OM_Response shall consist of MR_Response_Header and a Service_ResponsePDU.

Service_ResponsePDU shall be one of the following :

- Get_Attribute_Single_ResponsePDU
- Get_Attribute_All_ResponsePDU
- Get_Attribute_List_ResponsePDU
- Set_Attribute_Single_ResponsePDU
- Set_Attribute_All_ResponsePDU
- Set_Attribute_List_ResponsePDU
- Reset_ResponsePDU

- Create_ResponsePDU
- Delete_ResponsePDU
- Start_ResponsePDU
- Stop_ResponsePDU
- Find_Next_Object_Instance_ResponsePDU
- NOP_ResponsePDU
- Apply_Attributes_ResponsePDU
- Save_ResponsePDU
- Restore_ResponsePDU
- Group_Sync_ResponsePDU
- Add_AckData_Path_ResponsePDU
- Remove_AckData_Path_ResponsePDU
- Initialize_ResponsePDU
- Management_Message_ResponsePDU
- Connection_Bind_ResponsePDU
- Producing_Application_Lookup_ResponsePDU

CM_Service shall be either CM_Request or CM_Response.

CM_Request shall consist of MR_Request_Header and a CM_RequestPDU.

CM_RequestPDU shall be one of the following :

- Forward_Open_RequestPDU
- Forward_Close_RequestPDU
- Large_Forward _Open_RequestPDU
- Unconnected_Send_RequestPDU
- Get_Connection_Data_RequestPDU
- Search_Connection_Data _RequestPDU
- Get_Object_Owner_RequestPDU

CM_Response consists of MR_Response_Header and a CM_ResponsePDU.

CM_ ResponsePDU shall be one of the following :

- Forward_Open_ ResponsePDU
- Forward_Close_ ResponsePDU
- Large_Forward _Open_ ResponsePDU
- Unconnected_Send_ ResponsePDU
- Get_Connection_Data_ ResponsePDU
- Search_Connection_Data _ ResponsePDU
- Get_Object_Owner_ ResponsePDU

4.1.2.2 Transport_PDU structure

Transport_PDU is a sequence of a Transport_Header, followed by either OM_Service (Object Management ASE) or Application Data.

OM_Service is defined in 4.1.2.1 above.

Application Data comes from the source to which the connection has been made.

4.1.3 UCMM_PDUs

All UCMM_PDUs shall be sent and received using fixed tag 0x83 or Management tag 0x88. When a device becomes powered, the UCMM shall invoke the `DLL_enable_fixed_request` service of the data-link layer to request that fixed tag 0x83 or 0x88 Lpackets be routed to the UCMM. All sending and receiving of TPDUs shall use the `DLL_fixed_request.req` and `DLL_fixed_request.ind` service primitives of the data-link layer, respectively. The packet parameter of these services shall be prefixed with the following header to form the Transport PDU before sending to the data-link layer. The format of the UCMM_PDU Header is shown in Table 9.

Table 9 – UCMM_PDU header format

Parameter name	Format
command_code	USINT
Timeout	USINT
TransactionID	UINT

The `command_code` shall specify the type of UCMM_PDU as shown in Table 10. UCMM packets received with a reserved `command_code` shall be discarded without acknowledgement.

Table 10 – UCMM command codes

Command_code	Description
0	Reserved
1	acknowledge a request
2	request with retry until acknowledged
3	response with retry until acknowledged
4	request with no acknowledge and no response
5	acknowledge a response
6	response which shall not retry (no acknowledge)
7	request with retry until response (no acknowledge)
8	Request which shall not retry and shall cause a code 6 response
9 – 255	Reserved

The timeout field shall specify the duration of the transaction in an 8-bit floating-point format. The most significant 5 bits of the field shall be an unsigned exponent that is not biased. The least significant 3 bits shall be the least significant 3 bits of a 4 bit unsigned mantissa. The most significant bit of the mantissa shall be 1 and shall not appear explicitly in the 8-bit representation. The binary point shall be positioned between the implied 1 and the rest of the mantissa. The units of the transaction duration computed using the timeout field shall be in milliseconds.

NOTE Using ANSI C precedence rules, the number of 0,125 ms ticks is
`( 8 | timeout & 7 ) << ( timeout >> 3 & 31 )` .

The transactionID field shall be composed of two sub-fields:

- the least significant 10 bits, RECORD, shall identify a specific transaction;

- the most significant 6 bits, SEQUENCE, shall be used for duplicate detection. It shall be incremented for every unique message on this RECORD; however it shall not be incremented on a retry.

Only one message shall be outstanding for a given RECORD.

If multiple outstanding messages are required then multiple RECORDs are required.

When an I'm alive fixed tag packet is received from a node (see IEC 61158-4-2, Station Management), all UCMM transactions with that node shall be aborted.

4.1.4 Transport_Headers

4.1.4.1 General

Contents of Transport headers varies depending on the class of transport selected during the connection establishment.

4.1.4.2 Class 0 (null or base)

The class 0 transport header shall be an unsigned 16-bit number. This header shall be written to the TPDU buffer by the transport and shall be simply discarded by the transport from the packet coming from the consumer. The format of the Header is shown in Table 11.

Table 11 – Transport class 0 header

Parameter name	Format
don't_care	UINT

4.1.4.3 Class 1 (duplicate detection)

The class 1 transport header shall be an unsigned 16-bit sequence count. This header shall be written to the TPDU buffer by the transport and shall be read by the transport from the packet coming from the consumer. The format of the Header is shown in Table 12.

Table 12 – Transport class 1 header

Parameter name	Format
sequence_count	UINT

4.1.4.4 Class 2 (acknowledged)

Like the class 1 transport header, the class 2 transport header shall be a 16-bit sequence count. This header shall be written to the TPDU buffer by the transport and shall be read by the transport from the packet coming from the consumer. The format of the Header is shown in Table 13.

Table 13 – Transport class 2 header

Parameter name	Format
sequence_count	UINT

4.1.4.5 Class 3 (verified)

Like the class 1 transport header, the class 3 transport header shall be a 16-bit sequence count. This header shall be written to the TPDU buffer by the transport and shall be read by

the transport from the packet coming from the consumer. The format of the Header is shown in Table 14.

Table 14 – Transport class 3 header

Parameter name	Format
sequence_count	UINT

4.1.4.6 Class 4 (non-blocking), class 5 (non-blocking, fragmenting) and class 6 (multipoint connection, fragmenting)

These classes of transports all use the header format shown in Table 15. Classes 4 and 5 have two headers in each TPDU, one Sender and one Receiver. Class 6 uses a Sender header in the Client to Server direction and a Receiver header in the Server to Client direction.

Table 15 – Classes 4 to 6 header format

Command			Sequence number	Data	
MSB	Bits 15 – 12		Bits 11 – 8	Bits 7 – 0	LSB
Value:	Meaning in Sender Header:	Meaning in Receiver Header:	Increments from 0 to 15, then rolls over to zero again. Used in Sender header to distinguish a retry from a new transmission. Used in Receiver header to indicate which sender TPDU is being ACK'd or NAK'd.	For "First": Number of 256 octet blocks (minus 1) to allocate to hold complete message. For "NAK": Reason code. For others: Set to zero when generating; ignore when receiving.	
0.	Null	ACK			
1.	Only	NAK			
2.	First	Not Initialised			
3.	Middle	Undefined			
4.	Last	Undefined			
5. to 15 are undefined.					

The Command identifies the use of the TPDU by the transport. The Command field is defined differently for the Sender and the Receiver header.

Sender header commands :

Null = no data in this Transport PDU

Only = this is the Transport PDU conveying all of the Application data

First / Middle / Last = position of this Transport PDU within the segmented Application data

The Sequence Number is used in the Sender header to distinguish a retry from a new transmission. It is used in Receiver header to indicate which sender TPDU is being ACK'd or NAK'd. This is **NOT** used for applications request/response matching. This is only used by the transport state machines.

The Data is used by one Sender command and one Receiver command. It is used in the *First* Sender command to indicate the number of 256 octet blocks (minus 1) the Receiver shall allocate for the complete application message. That is, application messages of size 0 to 256 octets shall have the *First* Data field set to 0; 257 to 512, to 1; and so on up to 65 536 octets. It is used in the NAK Receiver command to provide a reason code:

Class 4 (the Point-to-Point non-Blocking transport) uses two headers in each TPDU, followed by the application data. The Sender Header is first in the TPDU, the Receiver Header is second, followed by the application data. Class 4 only uses the *Null* and *Only* Sender Commands (numbered 0 and 1).

Class 5 (the Point-to-Point Fragmenting non-Blocking transport) uses two headers in each TPDU, followed by the application data. The Sender Header is first in the TPDU, the Receiver Header is second, followed by the application data. Class 5 uses all of the Sender Commands.

Class 6 (the Multipoint connection Fragmenting transport) uses one header in each TPDU, followed by the application data. Class 6 uses all of the Sender Commands. The Sender Header is used in the Client-to-Server TPDU and the Receiver Header is used in the Server to Client TPDU.

4.1.4.7 Exclusive owner O⇒T data format

Exclusive owner connections shall have either of two real-time transfer formats:

- 32-bit header, fixed size;
- no header, variable size.

The 32-bit header prefixed to the real-time data shall be in the form shown in Table 16.

Table 16 – Real-time data header – exclusive owner

Parameter	Format	Size (bits)
Run_idle	RUN_IDLE	1
Reserved	UDINT	31

The `run_idle` flag (bit 0) shall be set (1 = `RUN`) to indicate that the following data shall be sent to the target application. It shall be clear (0 = `IDLE`) to indicate that the idle event shall be sent to the target application. The `reserved` field (bits 1-31) shall be reserved and set to 0.

If the no_header transfer format is used, the reception of a packet with data beyond the transport header shall indicate the RUN mode. If the packet is truncated after the transport header, the target shall be sent an idle event.

4.1.4.8 Redundant owner O⇒T data format

4.1.4.8.1 General format

Redundant owner shall have a 32-bit header prefixed to the real-time data. This header shall be in the form shown in Table 17.

Table 17 – Real-time data header– redundant owner

Parameter	Format	Size (bits)
Run_idle	RUN_IDLE	1
COO	BOOLEAN	1
ROO	USINT	2
Reserved	UDINT	28

4.1.4.8.2 Run_idle flag

The run_idle flag (bit 0) shall be have the same meaning as it does in an exclusive owner connection and shall be one of RUN or IDLE. The reserved field (bits 4-31) shall be reserved and set to 0.

4.1.4.8.3 Claim output ownership (COO) flag

The COO flag shall be set (1) when an originator application wants its connection to be the owning connection of the target application. The COO flag shall be reset (0) when an originator application does not want its connection to be the owning connection of the target application. When the owning connection resets (0) its COO flag, its sibling connections shall be checked for a set (1) COO flag. The new owner shall be any of the connections that have their COO flag set.

NOTE This results in undefined behaviour if more than one other connection has its COO flag set.

4.1.4.8.4 Ready for ownership of outputs (ROO) priority value

The ROO priority value shall be non-zero when an originator application does not want to force its connection to be the owning connection of the target application, but is ready to be the owning connection should there be no originator applications claiming to be the owning connection. The ROO priority value shall be zero when the originator application does not want to be the owning connection of the target connection and is not to be the owning connection should there be no originator application claiming to be the owning connection. The ROO priority value shall be used only when the COO flag is reset.

The value of the ROO field can range from 0 to 3. The originator applications shall each determine a unique non-zero ROO value.

4.1.4.8.5 Determining the owning connection

The originator applications shall determine among themselves which originator application has the owning connection. The owning connection shall be determined by the originator application that sets its COO flag. In situations where multiple originator applications have their COO flag set or where no originator connections have their COO flag set, the following rules shall be applied by the target transport to determine the owning connection.

- There shall be no owning connection until an originator application sends a real-time packet with the COO flag set;
- If there is only one originator application which had the COO flag set in its last real-time packet, that originator application shall have the owning connection;
- If there are multiple originator applications which had the COO flag set in its last real-time packet, the last originator application that transitioned its COO flag from reset to set shall have the owning connection.
- If the originator application with the owning connection resets its COO flag, closes its connection, or if that connection times out, and no other originator applications have their COO flags set, the originator application with the highest non-zero ROO priority value shall have the owning connection.
- If all of the originator applications have their COO flags reset and ROO priority values set to zero, there shall be no owning connection.
- When the first real-time packet containing a set COO flag is received by the target transport, the originator application that sent the real-time packet shall have the owning connection.

4.1.4.8.6 Transporting events and data to a target application

The connection related events from each of the redundant owner connections shall be combined using the following rules such that the target application sees only a single exclusive owner connection:

- Until an owning connection is initially determined, the transport shall not indicate real-time data reception to the target application;
- If an owning connection is determined, the transport shall indicate the event consistent with the owning connections real-time data to the target application. If the Run/Idle flag is reset in the real-time data for the owning connection, the transport shall indicate the idle state to the target application. If the Run/Idle flag is set in the real-time data for the owning connection, the transport shall indicate the run state and the real-time data to the target application;
- If an owning connection had been previously determined, but no originator is currently claiming or ready for ownership, the transport shall indicate idle state to the target application;
- If all the redundant connections are closed or have been timed out, the target transport shall indicate the event consistent with the last connection to have been closed or timed out to the target application.

4.1.5 CM_PDUs

4.1.5.1 General

Connection establishment and maintenance services are provided by the Connection Manager. They are

CM_Forward_Open (corresponds to Forward_Open and Ex_ Forward_Open PDUs)

CM_Forward_Close (corresponds to Forward_Close PDUs)

CM_Unconnected_Send (corresponds to Unconnected_Send PDUs)

CM_Get_Connection_Data (corresponds to Get_Connection_Data PDUs)

CM_Search_Connection_Data (corresponds to Search_Connection_Data PDUs)

CM_Get_Object_Owner (corresponds to Get_Object_ Owner PDUs)

4.1.5.2 Connection Manager

The Connection Manager object is used to manage the establishment and maintenance of communication connections.

The Connection Manager objects at different nodes shall communicate using the UCMM services described in IEC 61158-5-2. The TPDUs with which peer Connection Managers communicate shall be derived from the services of the Connection Manager.

4.1.5.3 Forward_Open

4.1.5.3.1 General

The response to the Forward_Open_Request_PDU is a Forward_Open_ResponsePDU. The Forward_Open response shall contain the original connection serial number and owner vendor and serial number and the connection IDs (CID) necessary to complete the connection if successful and error information if the connection failed.

NOTE 1 The Forward_Open request sets up network, transport, and application connections. An application connection consists of a single transport connection, and one or two network connections that are in turn comprised of multiple link connections. Each port segment in the connection path uses a link connection. The Forward_Open service between two devices builds one or two link connections as specified by the network connection parameter and the requested packet intervals (CM_RPI). Since up to two network connections can be required for a single transport connection, they are differentiated by the O⇒T and T⇒O designations; O⇒T means originator to target, and T⇒O means target to originator.

NOTE 2 The processing of a single Forward Open service may result in the creation of multiple active Connection object instances.

4.1.5.3.2 Format of Forward_Open request

The format of the Forward_Open request TPDU shall be of the form shown in Table 18.

Table 18 – Forward_Open request format

Parameter name	Format
priority_and_tick	SWORD
connection time-out ticks	USINT
O2T_CID	UDINT
T2O_CID	UDINT
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
connection timeout multiplier	USINT
reserved[3]	USINT
O2T_CM_RPI	UDINT
O2T_connection_parameters	UINT
T2O_CM_RPI	UDINT
T2O_connection_parameters	UINT
xport_type_and_trigger	SWORD
connection_path_size	USINT
connection_path	Padded EPATH

4.1.5.3.3 Format of Forward_Open response if success

If the `status` parameter of the `CM_open_response` is zero (no error), the format of the Forward_Open response TPDU shall be of the form shown in Table 19.

Table 19 – Forward_Open_Good response format

Parameter name	Format
O2T_CID	UDINT
T2O_CID	UDINT
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
O2T_CM_API	UDINT
T2O_CM_API	UDINT
Application_reply_size; (in 16-bit words)	USINT
Reserved	USINT
application_reply[]	UINT

Success shall be returned when the connection requested has been established from this point forward in the path. This response also shall indicate the connection serial number and the actual packet rate of the connection. Once the successful response has been received, the connection shall be open from this point forward in the path. Targets shall wait at least 10 seconds after sending the `CM_Forward_Open_Good_Response` for the first packet on a connection.

4.1.5.3.4 Format of Forward_Open response if failure

If the `status` parameter of the `CM_open_response` is not zero (error), the format of the Forward_Open response TPDU shall be of the form shown in Table 20.

Table 20 – Forward_Open_Bad response format

Parameter name	Format
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
remaining_path_size; (in 16-bit words)	USINT
Reserved	USINT

This format shall be used for all failures of the connection system. The requested connection shall not be established, and the object specific status words shall contain information about the reason for the failure. The `remaining_path_size` shall contain the length of the path at the point the connection failed. This information can be used to debug the problem.

In the failure response, the remaining `remaining_path_size` shall be the "pre-stripped" size. This shall be the size of the path when the node first receives the request and has not yet started processing it. A target node may return either the "pre-stripped" size or 0 for the remaining `remaining_path_size`.

A duplicate Forward_Open service shall be defined as a Forward_Open service whose `vendor_ID`, `connection_serial_number`, and `originator_serial_number` match an existing connection's parameters. If the duplicate Forward_Open service is a null Forward_Open service (defined as the connection type in both the O2T and T2O network connection parameter fields are NULL), then the Forward_Open service shall be forwarded to the application for further processing. Null Forward_Open requests may be used to reconfigure the connection. The Connection Manager in the intermediate nodes need not allocate additional resources for a duplicate Forward_Open request since the resources have already been allocated. If the duplicate Forward_Open request is not NULL, then an general status = 0x01, extended status = 0x0100 shall be returned.

The suggested originator behavior in the duplicate Forward_Open case should be to either close and re-establish the connection or wait for the connection to time-out and then establish the connection again. It shall be recognised that in the latter case, it shall take the connection 60 seconds (first data time-out) to time-out before the connection can be re-established.

4.1.5.4 Large_Forward_Open

4.1.5.4.1 General

The Large_Forward_Open shall have the same function as the Forward_Open except that it shall allow the establishment of connections larger than 511 octets.

The response to the Large_Forward_Open_Request_PDU is an Large_Forward_Open_ResponsePDU. The Large_Forward_Open response shall contain the original connection serial number and owner vendor and serial number and the connection IDs (CID) necessary to complete the connection if successful and error information if the connection failed.

NOTE The Large_Forward_Open request is very similar to the Forward_Open request.

4.1.5.4.2 Format of Large_Forward_Open request

The format of the Large_Forward_Open request TPDU shall be of the form shown in Table 21.

Table 21 – Large_Forward_Open request format

Parameter name	Format
priority_and_tick	SWORD
connection time-out ticks	USINT
O2T_CID	UDINT
T2O_CID	UDINT
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
connection timeout multiplier	USINT
reserved[3]	USINT
O2T_CM_RPI	UDINT
O2T_ex_connection_size	UINT
O2T_connection_parameters	UDINT
T2O_CM_RPI	UDINT
T2O_ex_connection_size	UINT
T2O_connection_parameters	UDINT
xport_type_and_trigger	SWORD
connection_path_size	USINT
connection_path	Padded EPATH

The connection size field in the two connection_parameters shall be reserved in this case and set to zero. The ex_connection_size parameters shall be used instead to specify the maximum size of the connection (up to 65 535).

4.1.5.4.3 Format of Large_Forward_Open response if success

If the `status` parameter of the `CM_open_response` is zero (no error), the format of the Large_Forward_Open response TPDU shall be of the form shown in Table 22.

Table 22 – Large_Forward_Open_Good response format

Parameter name	Format
O2T_CID	UDINT
T2O_CID	UDINT
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
O2T_CM_API	UDINT
T2O_CM_API	UDINT
Application_reply_size; (in 16-bit words)	USINT
Reserved	USINT
application_reply[]	UINT

Success shall be returned when the connection requested has been established from this point forward in the path. This response also shall indicate the connection serial number and the actual packet rate of the connection. Once the successful response has been received, the connection shall be open from this point forward in the path.

4.1.5.4.4 Format of Large_Forward_Open response if failure

If the `status` parameter of the `CM_open_response` is not zero (error), the format of the Large_Forward_Open response TPDU shall be of the form shown in Table 23.

Table 23 – Large_Forward_Open_Bad response format

Parameter name	Format
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
remaining_path_size; (in 16-bit words)	USINT
Reserved	USINT

Either a target device or an intermediate router shall return an Invalid Connection Size extended error code, along with the maximum connection size supported additional status word, if the connection size requested is not supported while processing a Large_Forward_Open request.

This format shall be used for all failures of the connection system. The requested connection shall not be established, and the object specific status words shall contain information about the reason for the failure. The `remaining_path_size` shall contain the length of the path at the point the connection failed. This information can be used to debug the problem.

In the failure response, the remaining `remaining_path_size` shall be the "pre-stripped" size. This shall be the size of the path when the node first receives the request and has not yet started processing it. A target node may return either the "pre-stripped" size or 0 for the remaining `remaining_path_size`.

The suggested originator behavior in the duplicate Forward_Open case should be to either close and re-establish the connection or wait for the connection to time-out and then establish the connection again. It shall be recognised that in the latter case, it shall take the connection 60 seconds (first data time-out) to time-out before the connection can be re-established.

4.1.5.5 Forward_Close

4.1.5.5.1 General

The Forward_Close request shall remove a connection from all the nodes participating in the original connection. The Forward_Close shall be sent between Connection Managers as specified in the connection_path. The Forward_Close request shall cause all resources in all nodes participating in the connection to be deallocated, including connection IDs, link transmit time, and internal memory buffers.

If an intermediate node cannot find the connection that is to be closed (it may have timed out at the node), the Forward_Close request shall still be forwarded to downstream nodes or the target application (via the `CM_close_indication`).

NOTE The Forward_Close is always forwarded to allow downstream nodes or the target application to release any resources that were allocated for the connection.

4.1.5.5.2 Format of Forward_Close request

The format of the Forward_Close request shall be of the form shown in Table 24.

Table 24 – Forward_Close request format

Parameter name	Format
connection_priority/tick time	SWORD
connection_timeout	USINT
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
connection_path_size	USINT
Reserved	USINT
connection_path	Padded EPATH

4.1.5.5.3 Format of Forward_Close response if success

The Forward_Close response shall be sent between Connection Managers in response to a Forward_Close request, and shall contain the status of the close request. If the `status` parameter of the `CM_close_response` is zero (no error), the format of the Forward_Close response TPDU shall be of the form shown in Table 25.

Table 25 – Forward_Close_Good response format

Parameter name	Format
connection serial number	UINT
vendor ID	UINT
originator serial number	UDINT
application_reply_size; in 16-bit words	USINT
Reserved	USINT
application_reply[]	UINT

A successful Forward_Close response shall be returned when the close has been acknowledged by all the nodes from this point in the path to the end of the path. The response shall indicate the `connection_serial_number`, `vendor_ID`, and `originator_serial_number` of the closed connection. Once the response indicating success has been received, the connection shall be closed from this point in the path to the end. The target application may optionally pass back application specific information in the successful close response. If no `application_reply` information is contained in the service the `application_reply_size` shall be zero.

4.1.5.5.4 Format of Forward_Close response if failure

If the `status` parameter of the `CM_close_response` is not zero (error), the format of the Forward_Close response shall be of the form shown in Table 26.

Table 26 – Forward_Close_Bad response format

Parameter name	Format
connection_serial_number	UINT
vendor_ID	UINT
originator_serial_number	UDINT
remaining_path_size	USINT
reserved	USINT

The object specific status words shall contain information about the reason for the failure. The `remaining_path_size` shall contain the path size from the point at which the close connection failed.

In the failure response, the `remaining_path_size` shall be the “pre-stripped” size. This shall be the size of the path when the node first receives the request and has not yet started processing it. A target node shall return either the “pre-stripped” size or 0 for the `remaining_path_size`.

4.1.5.6 Unconnected_Send

4.1.5.6.1 Format of Unconnected_Send request

The Unconnected_Send service shall allow an application to send a message to a device without first setting up a connection. The Unconnected_Send service shall use the Connection Manager object in each intermediate node to forward the message and to remember the return path. The UCMM of each link shall be used to forward the request from Connection Manager to Connection Manager just as it is for the Forward_Open service; however. no connection shall be built. The Unconnected_Send service shall be sent to the local Connection Manager and shall be sent between intermediate nodes. When an intermediate node removes the last port segment, the message shall be formatted as a UCMM message and sent to the port and link address of the last segment.

NOTE The target node never sees the Unconnected_Send service but only a standard message arriving via the UCMM.

If the message contains an odd number of octets, a pad octet shall be appended to the end of the message allowing the remaining fields to be aligned on a 16-bit boundary. The format of the Unconnected_Send request shall be of the form of the form shown in Table 27.

Table 27 – Unconnected_Send request format

Parameter name	Format
priority/tick time	SWORD
conn time-out ticks	USINT
Message_size; in octets; not including pad	UINT
Message Router PDU ; if message_router_PDU is odd size, include pad octet	MR_Request
pad	USINT
path_size	USINT
Reserved	USINT
path	Padded EPATH

The `message_size` shall be the size of the `message_router_PDU` in octets without the `pad`. The `reserved` field shall be set to zero. The `path_size` shall be the number of 16-bit words in the `path` field.

4.1.5.6.2 Unconnected_Send response

The Unconnected_Send response shall be generated by the last intermediate node from the UCMM response generated by the target node or by an intermediate node as the result of a UCMM time-out, a problem with the embedded message, or a problem with the Unconnected Service Request itself. The packet shall be routed from intermediate node to intermediate node using the information stored when the Unconnected_Send request was processed. The response shall contain a header with status information about the request and a variable length response generated by the target node.

4.1.5.6.3 Format of Unconnected_Send response if success

This format shall be used when an unconnected message response is received. The application response shall be the format of the Message response from the target and shall contain error codes that resulted from the execution of the message by the application at the target node. The structure of the response is shown in Table 28.

Table 28 – Unconnected_Send_Good response format

Parameter name	Format
application_reply [ ]	OCTET

4.1.5.6.4 Format of Unconnected_Send response if failure

This format shall be used if the originating node or any intermediate node found a problem in the message format, message parameters, or a UCMM timed-out due to no acknowledge or no response received. A UCMM time-out may be the result of

- the target device not being connected or powered up. (link specific time-out);
- an intermediate device not being connected or powered up. (link specific time-out);
- a target device being out of UCMM buffers or overloaded (Flow Control). (link specific time-out);
- an intermediate device being out of UCMM buffers or overloaded (Flow Control). (link specific time-out);
- an invalid path segment type (logical or network segment);
- an invalid port segment address (link or port address invalid).

Since the Unconnected message system does not implement any flow control, the failure of an Unconnected_Send message need not mean a device is not present in the system. The format of the Unconnected_Send response service for a failure shall be of the form shown in Table 29.

Table 29 – Unconnected_Send_Bad response format

Parameter name	Format
remaining_path_size	USINT
pad	USINT

In the failure response, the remaining Connection Path Size shall be the “pre-stripped” size. This shall be the size of the path when the node first receives the request and has not yet started processing it. A target node may return either the ”pre-stripped” size or 0 for the remaining Connection Path Size.

4.1.5.7 Get_Connection_Data

4.1.5.7.1 General

This service shall return the parameters associated with a specified connection number. The connection number may be different from device to device even for the same connection. The connection number corresponds to the offset into the Connection Manager attribute that enumerates the status of the connections.

NOTE This service may be used for network diagnostics.

4.1.5.7.2 Format of Get_Connection_Data request

The format of the Get_Connection_Data request shall be of the form shown in Table 30.

Table 30 – Get_Connection_Data request format

Parameter name	Format
connection_number	UINT

4.1.5.7.3 Format of Get_Connection_Data response

The format of the Get_Connection_Data response TPDU shall be of the form shown in Table 31.

Table 31 – Get_Connection_Data response format

Parameter name	Format
connection_number	UINT
connection_state	UINT
originator_port	UINT
target_port	UINT
connection_serial_number	UINT
vendor_ID	UINT
serial_number	UDINT
originator_O2T_CID	UDINT
target_O2T _CID	UDINT
O2T_connection timeout multiplier	USINT
reserved1[3]	USINT
originator_O2T_CM_RPI	UDINT
originator_O2T_CM_API	UDINT
originator_T2O_CID	UDINT
target_ T2O_CID	UDINT
T2O_connection timeout multiplier	USINT
reserved2[3]	USINT
originator_T2O_CM_RPI	UDINT
originator_T2O_CM_API	UDINT

4.1.5.8 Search_Connection_Data

4.1.5.8.1 General

This service shall return the parameters associated with a specified connection within a device identified by vendor and serial number .

NOTE This service may be used for network diagnostics.

4.1.5.8.2 Format of Search _Connection_Data request

The format of the Search_Connection_Data request shall be of the form shown in Table 32.

Table 32 – Search_Connection_Data request format

Parameter name	Format
connection_serial_number	UINT
vendor_ID	UINT
serial_number	UDINT

4.1.5.8.3 Format of Search _Connection_Data response

The format of the Search_Connection_Data response shall be the same as the response from the Get_Connection_Data service (see 4.1.5.7).

4.1.5.9 Get_Object_Owner

4.1.5.9.1 General

The Get_Object_Owner service of the Connection Manager shall return data about the connection(s) that own(s) a particular object. It shall be implemented in any device that accepts redundant connections.

4.1.5.9.2 Format of Get_Object_Owner request

The format of the Get_Object_Owner request TPDU shall be of the form shown in Table 33:

Table 33 – Get_Object_Owner request format

Parameter name	Format
reserved	USINT
path_size	USINT
path[]	UINT

The reserved field shall be set to zero.

4.1.5.9.3 Format of Get_Object_Owner response

The format of the Get_Object_Owner response TPDU shall be of the form shown in Table 34.

Table 34 – Forward_Open_Good response format

Parameter name	Format
number_of_connections	USINT
number_claiming_ownership	USINT
number_ready_for_ownership	USINT
last_action	USINT
connection_serial_number	UINT
originator_vendor_ID	UINT
originator_serial_number	UDINT

4.1.6 CM PDU components

4.1.6.1 Network connection parameters

4.1.6.1.1 Format

Network connection parameters shall be provided as a single 16-bit word that contains five fields, shown in Figure 4. The fields within the 16-bit word shall indicate

- the type of network connection desired;
- whether the buffer size is variable or fixed;
- the priority of the connection;
- the size of the connection buffer required.

The size of the connection buffer shall include any transport header.

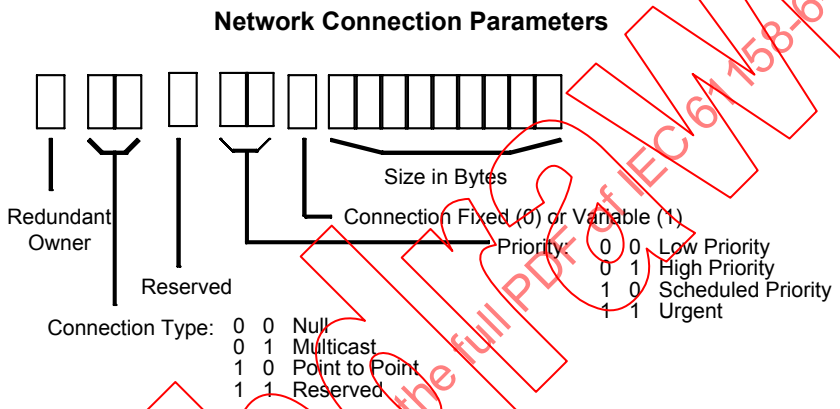


Figure 4 – Network connection parameters

4.1.6.1.2 Connection type

The connection type shall be one of NULL, MULTIPOINT, or POINT2POINT. The NULL type shall indicate that no network connection is required, while MULTIPOINT and POINT2POINT refer to the network connection types as defined in IEC 61158-5-2. The NULL connection type may be used to reconfigure a connection.

4.1.6.1.3 Priority

Priority shall be one of `LOW`, `HIGH`, and `SCHEDULED` as specified in IEC 61158-4-2.

NOTE `SCHEDULED` data is prioritised on a link wide basis; however, the two unscheduled priorities (`LOW` and `HIGH`) are arbitrated independently at each node. One node may transmit a `LOW` priority packet while another node has `HIGH` priority data to send.

4.1.6.1.4 Connection fixed/variable

The connection can be set up to use fixed or variable sized connections. With a fixed size connection, each transmission on the connection shall use the same size buffer. Otherwise, either a buffer overrun or underrun condition shall occur. With a variable sized connection, each transmission on the connection may send a variable amount of data up to the maximum size, which shall be specified when the connection is opened. Buffer underruns shall not be reported, but a buffer overrun can still occur if too much data is sent.

4.1.6.1.5 Connection size

The network connection size shall be the size of the buffer required for the connection, which includes the data and any transport header. For a variable sized connection, the size shall be the maximum size of the buffer for any transfer. The actual size of the transfer for a variable

connection shall be equal to or less than the size specified for the network connection. The maximum buffer size shall be dependent on the links that the connection traverses.

4.1.6.1.6 Redundant owner

The redundant owner bit in the O⇒T direction shall be set (= 1) to indicate that more than one owner may be permitted to make a connection simultaneously. The bit shall be clear (= 0) to indicate an exclusive-owner, input only or listen-only connection.

4.1.6.1.7 Reserved parameters

Reserved bits shall be clear (= 0).

4.1.6.2 Packet interval

4.1.6.2.1 Requested packet interval (CM_RPI)

The requested packet interval shall be the requested time between packets in microseconds. The format of the CM_RPI shall be a 32-bit integer in microseconds.

4.1.6.2.2 Actual packet interval (CM_API)

The actual packet interval shall be the actual time between packets in microseconds. The format of the CM_API shall be a 32-bit integer in microseconds.

4.1.6.2.3 Usage

The requested packet interval shall be the time between packets requested by the receiving device. The value shall be used to allocate link transmit time at each of the producing nodes. The allocation of link transmit time may have to be adjusted when the actual packet rate or actual packet interval is returned, since it is possible for the two values to differ. The path time-out value at each of the intermediate and target nodes shall also be set to the connection time-out multiplier times the CM_API. The CM_RPI is therefore required for all connections.

4.1.6.2.4 Function of priorities

Scheduled Priority

For scheduled priority, the CM_RPI shall be the packet rate of the repetitive data. On links that support allocation of link transmit time, link transmit time shall be reserved for this packet. For scheduled priority, the data shall also be restricted to the specified packet rate, which means that if data arrives at an intermediate node faster than the specified packet rate, the node shall filter the packets to the specified rate. Since each node's scheduled priority update rate is in discrete quanta, the Actual Packet Interval (CM_API) may be smaller (more rapid) than the CM_RPI. The path time-out value shall be set to the Connection Time-out multiplier times the CM_API.

High Priority

For high priority, the CM_RPI shall be used to set the path time-out in the intermediate and target nodes. The CM_RPI shall therefore be set to the slowest packet rate expected, which shall preclude having the connection close due to a path time-out. The longer the path time-out, the longer the time required to reclaim resources in the intermediate nodes as a result of faults in the network. Since the high priority is not quantised at any of the nodes, the CM_API shall equal the CM_RPI. To maintain consistency, however, the time-out value shall again be set to the Connection Time-out multiplier times the CM_API.

Low Priority

For low priority, the CM_RPI shall be used to set the path time-out in the intermediate and target nodes. The CM_RPI shall therefore be set to the slowest packet rate expected, which shall preclude having the connection close due to a path time-out. The longer the path time-

out, the longer the time required to reclaim resources in the intermediate nodes as a result of faults in the network. Since the low priority is not quantised at any of the nodes, the CM_API shall equal the CM_RPI. To maintain consistency, however, the time-out value shall again be set to the Connection Time-out multiplier times the CM_API.

4.1.6.3 Connection time-out multiplier

The Connection Time-out Multiplier specifies the multiplier applied to the CM_RPI to obtain the connection time-out value. Device shall stop transmitting on a connection whenever the connection times out even if the pending close has been sent. The multiplier shall be as represented by Table 35.

Table 35 – Time-out multiplier

Value	Multiplier	Notes
0	X 4	Default
1	X 8	
2	X 16	
3	X 32	
4	X 64	
5	X 128	
6	X 256	
7	X 512	
8 – 255	reserved	

4.1.6.4 Connection request priority and tick time

Two values are packed into this field. The connection request priority determines the priority of the UCMM messages used to establish the connection and has no correlation with the priority of the connection being made. Although Low priority is used for most connection establishments, High priority may be used for those special circumstances which require a connection be made quickly. The use of low priority was chosen to avoid congestion with time-critical messaging. UCMM messages can not use scheduled priority.

The tick time shall be used in conjunction with the connection request time-out ticks value. This value determines the time between ”ticks” in the latter field. Thus, if the tick time is 1ms., then a connection request time-out value of 5 would translate into 5 ms. If the tick time was 2 ms, then a connection request time-out value of 5 would translate into 10 ms.

The format of the Connection Request Priority/Tick Time shall be as shown in Figure 5.

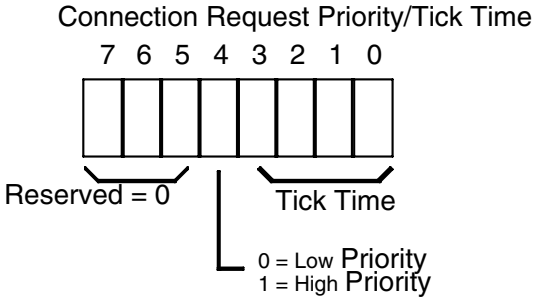


Figure 5 – Time tick

Table 36 shall determine the time between ticks.

Table 36 – Time tick units

Tick time		
Tick time (binary)	**Time per tick**	**Max. time**
0000	1 ms	255 ms
0001	2 ms	510 ms
0010	4 ms	1 020 ms
0011	8 ms	2 040 ms
0100	16 ms	4 080 ms
0101	32 ms	8 160 ms
0110	64 ms	16 320 ms
0111	128 ms	32 640 ms
1000	256 ms	65 280 ms
1001	512 ms	130 560 ms
1010	1 024 ms	261 120 ms
1011	2 048 ms	522 240 ms
1100	4 096 ms	1 044 480 ms
1101	8 192 ms	2 088 960 ms
1110	16 384 ms	4 177 920 ms
1111	32 768 ms	8 355 840 ms

4.1.6.5 Connection request time-out ticks

The connection request time-out ticks shall be used to specify the amount of time the originating application shall wait for the connection to be established. Each hop of the connection has a link specific time-out value implemented by the UCMM, and after the automatic retries have been exhausted without an acknowledge the UCMM shall generate a time-out error. Therefore if a device in the path shall be either not present or too busy to acknowledge a UCMM request, a UCMM time-out shall occur as fast as would be possible. The only time the connection request time-out period would be encountered shall be if the target node acknowledges the connection request but fails to respond or if the connection request shall be delivered but a failure in the network prevents the response from being returned. The connection request time-out shall be not the path time-out value derived from the CM_RPI and used to close a connection after it shall be established. The connection request time-out shall be specified by a combination of the connection request time-out ticks multiplied by the time per tick which shall be determined by the tick time value.

Since a particular connection request time-out can be expressed in several different formats, the rule shall be to **always pack it such that the tick time value shall be minimised.** This allows the best granularity for decreasing the time-out as the message is passed through hops. This also means that the tick time value may change as the message is passed through multiple hops. As an example, 300 ms may be encoded as:

1. 0001 10010110 -> Time per Tick = 2 ms, Ticks = 150 (right)

2. 0010 01001011 -> Time per Tick = 4 ms, Ticks = 75 (wrong)

The connection request time-out value shall be dependent on the number of port segments in the connection path, the processing time at the target node, and the congestion of the intermediate networks. The connection request time-out can be a relatively large value since the full connection request time-out shall only occur in a few rare cases. The only time the full connection request time-out would occur would be if:

- the connection request was successfully delivered to the target node and the node failed to generate a response. This would usually be caused by a firmware error in the device.
- the connection request was successfully delivered to the target node and the response was discarded due to congestion at an intermediate node.
- the connection request was successfully delivered to the target node and the target node or one of the intermediate nodes was removed or powered down before the response was returned.

The UCMM shall be used to forward the connection request to the next node in the path. The UCMM sends the request and then waits a link specific time for an acknowledgement or response, if no acknowledgement or response shall be received before the link specific time-out, the request shall be retried. After a link specific number of retries (typically three), the UCMM shall signal a time-out on the connection request. The Connection Manager which generated the request shall return an error response to the connection request. Therefore the connection request shall time-out as soon as possible for the most common system faults

- missing or powered down devices in the path;
- congestion at a device in the path.

To aid in debugging systems, each hop in the path shall decrement the time-out by about 80 ms before sending it out in the Forward_Open to the next device in the path. Each intermediate device (router) shall round the connection time-out value that shall be decremented such that a minimum of 1 tick shall be subtracted. This shall enable the last device before the failure to time-out first and return information about how far the connection was made before the failure. This information can be used to help in debugging problems in a system. A typical system function is shown in Figure 6.

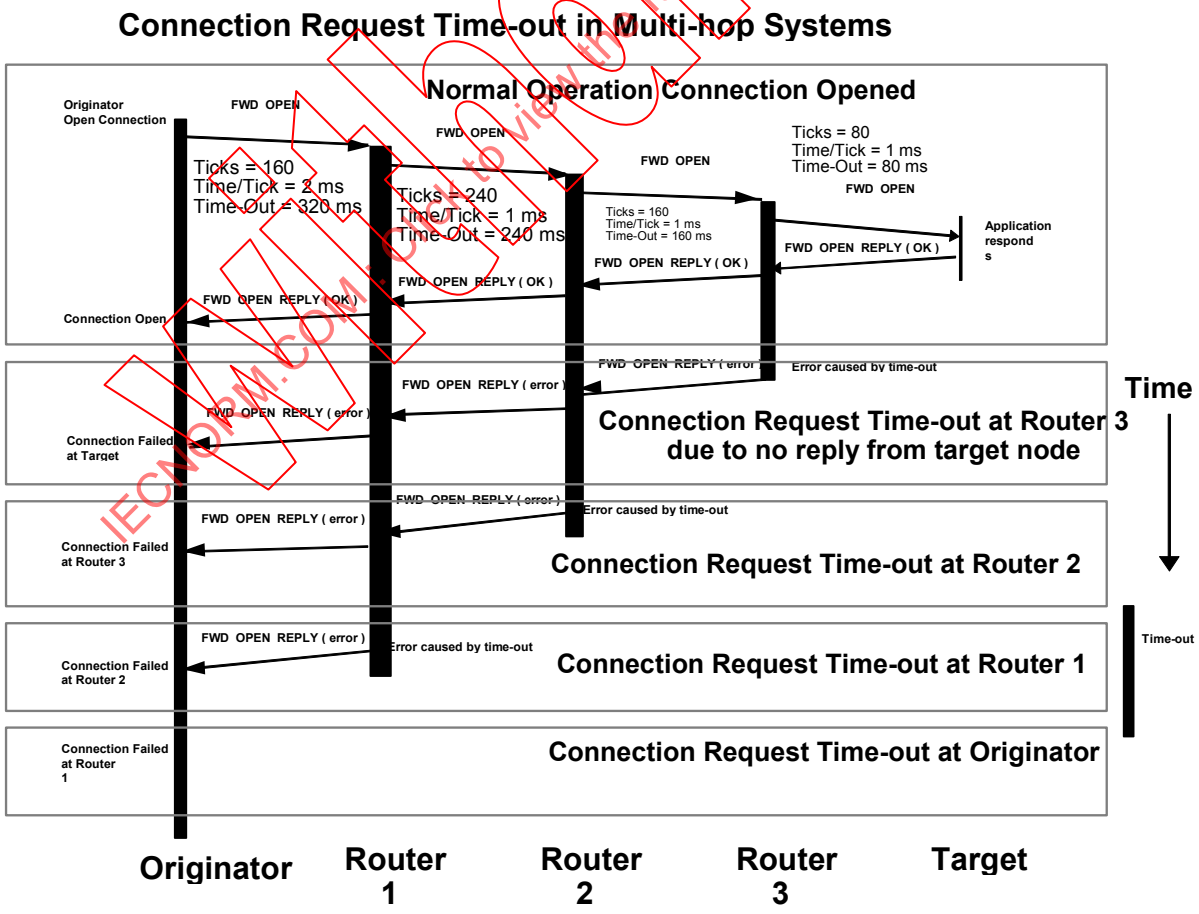


Figure 6 – Connection establishment time-out

In Figure 6 a connection is requested over a path containing 4 hops. The Connection request Time-out is specified as 320 ms by the originator of the connection. In the first case the connection is completed and the calculated time-out values are shown at each hop in the path. The second case is where the last device receives an acknowledgement but does not receive a response and times out. It returns a time-out error to the originator. Since the time-out value decreases for each hop, the time-out error shall be returned to the originator before the originator times out.

The other cases show time-outs at the other devices, and in each case the time-out error shall be returned to the originator before the originator itself times out. The last case is where no response is received by the originator and the connection open process is timed out. The reason to decrease the time-out at each node is so additional error information indicating where in the path the time-out occurred can be sent. Remember that if any of the devices do not receive an acknowledgement, the connection request shall immediately be timed out and an error response shall be generated.

4.1.6.6 Connection serial number

The connection serial number shall be a unique 16-bit value selected by the Connection Manager at the originator of the connection. The originator shall make sure that the 16-bit value is unique for the device. There shall be no other significance placed on the number by any other nodes in the connection path. The connection serial numbers shall be unique but do not have to be sequential. For example, an operator interface may have a large number of connections open at the same time, each with a unique number. The same values could be repeated at other operator interface stations. A possible implementation would be to have a connection list which points to the descriptor for each connection, and the connection serial number could be the index into the table.

4.1.6.7 Vendor ID

The vendor number shall be a unique number assigned to the various vendors of products. Each vendor has a unique number assigned.

4.1.6.8 Originator serial number

The originator serial number shall be a unique 32-bit value that is assigned to a device at the time of manufacture. This value shall be guaranteed to be unique for all devices manufactured by the same vendor. No significance shall be attached to the number. The combination of Vendor ID and Originator Serial Number shall be unique throughout the entire system.

4.1.6.9 Connection number

The connection number shall be a 16-bit value that is assigned by the Connection Manager when a connection is opened. This value allows other nodes to obtain connection data from the Connection Manager. This number shall not be confused with the Connection Serial Number.

4.1.6.10 Connection path size

The connection path size shall be the length of the connection path in 16-bit words. The length of the connection path varies during the connection process, since each node in the connection path removes the current port segment and forwards only the remaining path segments to the next node.

4.1.6.11 Connection path

Refer to Path description in 4.1.8.8.

4.1.6.12 Connection remaining path

Refer to Path description in 4.1.8.8.

4.1.6.13 Network connection ID

The Network Connection ID shall be a 32-bit value built as follows :

- Bits 31-24 shall be reserved and shall be set to zero;
- Bits 23-16 shall contain a node MAC ID depending on selected connection type
- Bits 15-0 shall contain the 16-bit connection number.

Table 37 shows options for selection of Connection ID.

Table 37 – Selection of connection ID

Connection type	MAC ID	Connection number
Multipoint	MAC of Producer	Unique by Device
Point to Point	MAC of Consumer	Unique by Device

The Network Connection ID shall be link specific and shall not be related to the connection serial number, which is connection specific and the same over all the links. The fields of the Network Connection ID shall be used to set the screening mechanism for the specified link.

The generic tag for the Lpackets exchanged over a connection and described in IEC 61158-4-2, shall be built as follows, based on the contents of the CID fields:

- the first transmitted octet of the generic tag shall contain the MAC ID;
- the second transmitted octet of the generic tag shall contain the least significant octet of the connection number;
- the third transmitted octet of the generic tag shall contain the most significant octet of the connection number.

To guarantee that connection IDs are unique on a link the following rules shall be used when choosing CIDs:

- the producer shall choose the CID for a multipoint transport;
- the consumer shall choose the CID for a point-to-point transport;
- on a CP 2/1 link, the node responsible for the choice of the CID (either producer or consumer) shall insert its MAC ID into bits 23-16 of the CID in the Forward_Open;
- a multipoint CID shall not be reused until all connections associated with the CID have been closed or timed out;
- receiving an I'm alive fixed tag packet from a node shall immediately close any connections with that device.

4.1.6.14 Transport class and trigger

The transport class and trigger specify the type of transport required for the connection. This information shall not be used by the Connection Manager but passed on to the application. The application shall determine if the transport type is supported and if an instance of the required transport is available. The octet that encode the transport class and trigger shall be of the following form shown in Table 38.

Table 38 – Transport class, trigger and Is_Server format

TransportClass (4 bits)	TriggerMode (3 bits)	Is_Server (1 bit)
NULL = 0 DUPLICATE_DETECT = 1 ACKNOWLEDGED = 2 VERIFIED = 3 NONBLOCKING = 4 FRAGMENTING = 5 MULTIPOINT_FRAG = 6	CYCLIC = 0 CHANGE_OF_STATE = 1 APPLICATION = 2	IS_NOT_SERVER = 0 IS_SERVER = 1

The least significant 4 bits, `class`, shall determine which transport type is specified and shall be in the range 0 to 6. The `class` values 7 through 15 shall be reserved. Bits 4-6, called `trigger`, shall determine what event causes the production of data on the connection and shall be in the range 0 to 2. The `trigger` values 3 through 7 shall be reserved. For transports that need to specify the target as a server, the `is_server` field shall be 1; otherwise, it shall be 0. For transport classes 0 and 1, the `is_server` field shall be ignored. Bit 7 shall be the `is_server` field;

See IEC 61158-5-2 for details on the behaviour for the different combinations of `is_server`, `class`, and `trigger`, CM_RPI, and CM_API.

4.1.7 MR headers

All request packets delivered to the Message Router, either from a connection or from the UCMM, shall have a header of the form shown in Table 39.

Table 39 – MR_Request_Header format

Parameter	Format
Service	USINT
path_size	USINT
path[]	UINT

The first octet, service, shall be directly delivered to the application object and shall be in the range 1 through 127. The `path_size` shall determine the number of 16-bit words in the `path` array. The `path` shall contain addressing information for the packet and shall be padded.

Response packets from the Message Router shall have a header of the form shown in Table 40.

Table 40 – MR_Response_Header format

Parameter	Format
Service	USINT
Reserved	USINT
general_status	USINT
Extended_status_size	USINT
Extended_status[]	WORD

The `service` field shall be the value of the `MR_Request_Header.service` field with its most significant bit set. If `MR_Request_Header.service` equals 0x05, the `MR_Response_Header.service` shall equal 0x85. The `general_status` field shall be filled using the `Status Code` parameter of the service response. Likewise the `extended_status`, `extended_status_size`, and `response` shall be filled using the `Extended Status` and other response parameters from the service response, respectively.

The `response_size` shall not be explicitly included in the `MR_Response_Header`; it shall be implied by number of octets in the `MR_Response_Header`.

4.1.8 OM_Service_PDU

4.1.8.1 General syntax

4.1.8.1.1 Get_Attribute_All

The Get_Attribute_All_RequestPDU body is empty.

The Get_Attribute_All_ResponsePDU body shall be as specified in Table 41.

Table 41 – Structure of Get_Attribute_All_ResponsePDU body

Name	Data type
Attribute Data	Object/class attribute-specific

4.1.8.1.2 Set_Attribute_All

The Set_Attribute_All_RequestPDU body shall be as specified in Table 42.

Table 42 – Structure of Set_Attribute_All_RequestPDU body

Name	Data type
Attribute Data	Object/class attribute-specific

The Get_Attribute_All_ResponsePDU body is empty.

4.1.8.1.3 Get_Attribute_List

The Get_Attribute_List_RequestPDU body shall be as specified in Table 43.

Table 43 – Structure of Get_Attribute_List_RequestPDU body

Name	Data type
Attribute Count	UINT
Attribute List	Array of UINT's

The Get_Attribute_List_ResponsePDU body shall be as specified in Table 44.

Table 44 – Structure of Get_Attribute_List_ResponsePDU body

Name	Data type	Semantics of values
Attribute Count	UINT	
Attribute Data	Array of STRUCT	
Attribute Identifier	UINT	
Attribute Status	UINT	Values 0-255 are reserved to mirror common service status codes. Values 256-65 535 are available for object/class attribute-specific errors.
Attribute Value	Object/class attribute- specific	

4.1.8.1.4 Set_Attribute_List

The Set_Attribute_List_RequestPDU body shall be as specified in Table 45.

Table 45 – Structure of Set_Attribute_List_RequestPDU body

Name	Data type
Attribute Count	UINT
Attribute Data	Array of STRUCT
Attribute Identifier	UINT
Attribute Value	Object/class attribute- specific

The Set_Attribute_List_ResponsePDU body shall be as specified in Table 46.

Table 46 – Structure of Set_Attribute_List_ResponsePDU body

Name	Data type	Semantics of values
Attribute Count	UINT	
Attribute Status List	Array of STRUCT	
Attribute Identifier	UINT	
Attribute Status	UINT	Values 0-255 are reserved to mirror common service status codes. Values 256-65 535 are available for object/class attribute-specific errors

4.1.8.1.5 Reset

The Reset_RequestPDU body shall be as specified in Table 47, if the **optional** "Object Specific Data" service parameter of the Reset request is specified. Else it shall be empty.

Table 47 – Structure of Reset_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Reset_ResponsePDU body shall be as specified in Table 48, if the **optional** "Object Specific Data" service parameter of the Reset response is specified. Else it shall be empty.

Table 48 – Structure of Reset_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.6 Start

The Start_RequestPDU body shall be as specified in Table 49, if the **optional** "Object Specific Data" service parameter of the Start request is specified. Else it shall be empty.

Table 49 – Structure of Start_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Start_ResponsePDU body shall be as specified in Table 50, if the **optional** "Object Specific Data" service parameter of the Start response is specified. Else it shall be empty.

Table 50 – Structure of Start_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.7 Stop

The Stop_RequestPDU body shall be as specified in Table 51, if the **optional** "Object Specific Data" service parameter of the Stop request is specified. Else it shall be empty.

Table 51 – Structure of Stop_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Stop_ResponsePDU body shall be as specified in Table 52, if the **optional** "Object Specific Data" service parameter of the Stop response is specified. Else it shall be empty.

Table 52 – Structure of Stop_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.8 Create

The Create_RequestPDU body shall be as specified in Table 53, if the **optional** "Object Specific Data" service parameter of the Create request is specified. Else it shall be empty.

Table 53 – Structure of Create_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Create_ResponsePDU body shall be as specified in Table 54, if the **optional** "Object Specific Data" service parameter of the Create response is specified. Else it shall be empty.

Table 54 – Structure of Create_ResponsePDU body

Name	Data type
Instance ID	UINT
Object Specific Data	Object/class service- specific

4.1.8.1.9 Delete

The Delete_RequestPDU body shall be as specified in Table 55, if the **optional** "Object Specific Data" service parameter of the Delete request is specified. Else it shall be empty.

Table 55 – Structure of Delete_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Delete_ResponsePDU body shall be as specified in Table 56, if the **optional** "Object Specific Data" service parameter of the Delete response is specified. Else it shall be empty.

Table 56 – Structure of Delete_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.10 Get_Attribute_Single

The Get_Attribute_Single_RequestPDU body is empty.

The Get_Attribute_ Single_ResponsePDU body shall be as specified in Table 57.

Table 57 – Structure of Get_Attribute_Single_ResponsePDU body

Name	Data type
Attribute Data	Object/class attribute-specific

4.1.8.1.11 Set_Attribute single

The Set_Attribute_Single_RequestPDU body shall be as specified in Table 58.

Table 58 – Structure of Set_Attribute_Single_RequestPDU body

Name	Data type
Attribute Data	Object/class attribute-specific

The Set_Attribute_Single_ResponsePDU body shall be as specified in Table 59, if the **optional** "Object Specific Data" service parameter of the Set_Attribute_Single response is specified. Else it shall be empty.

Table 59 – Structure of Set_Attribute_Single_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.12 Find_Next_Object_Instance

The Find_Next_Object_Instance_RequestPDU body shall be as specified in Table 60.

Table 60 – Structure of Find_Next_Object_Instance_RequestPDU body

Name	Data type
Maximum Returned Values	USINT

The Find_Next_Object_Instance_ResponsePDU body shall be as specified in Table 61.

Table 61 – Structure of Find_Next_Object_Instance_ResponsePDU body

Name	Data type
Number Of List Members	USINT
Instance ID List	Array of UINT

4.1.8.1.13 NOP

The NOP_RequestPDU body is empty.

The NOP_ResponsePDU body is empty.

4.1.8.1.14 Apply_Attributes

The Apply_Attributes_RequestPDU body shall be as specified in Table 62, if the **optional** "Object Specific Data" service parameter of the Apply_Attributes request is specified. Else it shall be empty.

Table 62 – Structure of Apply_Attributes_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Apply_Attributes_ResponsePDU body shall be as specified in Table 63, if the **optional** "Object Specific Data" service parameter of the Apply_Attributes response is specified. Else it shall be empty.

Table 63 – Structure of Apply_Attributes_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.15 Save

The Save_RequestPDU body shall be as specified in Table 64, if the **optional** "Object Specific Data" service parameter of the Save request is specified. Else it shall be empty.

Table 64 – Structure of Save_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Save_ResponsePDU body shall be as specified in Table 65, if the **optional** "Object Specific Data" service parameter of the Save response is specified. Else it shall be empty.

Table 65 – Structure of Save_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.16 Restore

The Restore_RequestPDU body shall be as specified in Table 66, if the **optional** "Object Specific Data" service parameter of the Restorerequest is specified. Else it shall be empty.

Table 66 – Structure of Restore_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Restore_ResponsePDU body shall be as specified in Table 67, if the **optional** "Object Specific Data" service parameter of the Restore response is specified. Else it shall be empty.

Table 67 – Structure of Restore_ResponsePDU body

Name	Data type
Object Specific Data	Object/class service- specific

4.1.8.1.17 Group_Sync

The Group_Sync_RequestPDU body shall be as specified in Table 68.

Table 68 – Structure of Group_Sync_RequestPDU body

Name	Data type
Object Specific Data	Object/class service- specific

The Group_Sync_ResponsePDU body shall be as specified in Table 69.

Table 69 – Structure of Group_Sync_ResponsePDU body

Name	Data type	Semantics of values
IsSynchronized	BOOL	Indicates if object is synchronized to the PTP Time Master 0 = Not synchronized 1 = Is synchronized
Object Specific Data	Object/class service- specific	Object/class service- specific

4.1.8.2 Identity object specific syntax elements

4.1.8.2.1 Attributes

4.1.8.2.1.1 Class attributes

The format of the Identity object class attributes shall be as specified in Table 70.

Table 70 – Identity object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 1
2	Max Instance	UINT	
6	Max ID Number of Class Attributes	UINT	
7	Max ID Number of Instance Attributes	UINT	

4.1.8.2.1.2 Instance attributes

The format of the Identity object instance attributes is as specified in Table 71.

Table 71 – Identity object instance attributes

Attribute ID	Name	Data type	Semantics of values
1	Vendor ID	UINT	The value zero shall not be used.
2	Device Type	UINT	A listing of the Device Type ranges can be found in 4.1.10.2.5.
3	Product Code	UINT	The value zero is not valid.
4	Revision	STRUCT of	The value zero shall not be valid for either the Major and Minor Revision fields of a compliant product. A damaged product may return zero to indicate unknown revision if its storage of revision become corrupt.
	Major Revision	USINT	The Major Revision attribute shall be limited to 7 bits. The eighth bit is reserved and shall be zero.
	Minor Revision	USINT	
5	Status	WORD	Bit definitions are described in Table 72.
6	Serial Number	UDINT	
7	Product Name	SHORT_STRING	The maximum number of 8-bit characters in this string shall be 32. Each of the characters shall be in the range 0x20 to 0x7E.
8	State	USINT	Present state of the device as represented by the state transition diagram: 0 = Nonexistent 1 = Device Self Testing 2 = Standby 3 = Operational 4 = Major Recoverable Fault 5 = Major Unrecoverable Fault 6 – 254 = Reserved 255 = Default for Get_Attributes_All service
9	Configuration Consistency Value	UINT	Contents identify configuration of device
10	Heartbeat Interval	USINT	The nominal interval between heartbeat messages in seconds.
11	Active Language	STRUCT of	Currently active language for the device.
	Language1	USINT	The language1 field from the STRINGI data type.
	Language2	USINT	The language2 field from the STRINGI data type.
	Language3	USINT	The language3 field from the STRINGI data type.
12	Supported Language	ARRAY of STRUCT of	List of languages supported by character strings of data type STRINGI within the device.
	Language1	USINT	The language1 field from the STRINGI data type.
	Language2	USINT	The language2 field from the STRINGI data type.
	Language3	USINT	The language3 field from the STRINGI data type.
13	International Product Name	STRINGI	Names of the product in various languages
14	Semaphore	STRUCT of	Access Synchronization Semaphore
	Vendor Number	UINT	Default: All zero values
	Client Serial Number	UDINT	
	Semaphore Timer	ITIME	Timer valid range 100 thru 32 767
15	Assigned_Name	STRING1	Use assigned name
16	Assigned_Description	STRING1	User assigned description
17	Geographic_Location	STRING1	User assigned location

Table 72 – Identity object bit definitions for status instance attribute

Bit(s)	Called	Definition
0	Owned	1 for TRUE, 0 for FALSE
1		Reserved, set to 0
2	Configured	1 for TRUE, 0 for FALSE
3		Reserved, set to 0
4,5,6,7		See Table 73
8	Minor Recoverable Fault	1 for TRUE, 0 for FALSE
9	Minor Unrecoverable Fault	1 for TRUE, 0 for FALSE
10	Major Recoverable Fault	1 for TRUE, 0 for FALSE
11	Major Unrecoverable Fault	1 for TRUE, 0 for FALSE
12, 13		Reserved, shall be set to 0
14, 15		Reserved, shall be set to 0

The values of the status bits 4 to 7 shall be as shown in Table 73.

Table 73 – Bits 4 – 7 of status instance attribute

Bits 4, 5, 6, 7:	Meaning
0 0 0 0	Self–Testing (power–up) or State Unknown
0 0 0 1	Firmware Update in progress
0 0 1 0	Communication Fault (lost connection)
0 0 1 1	Unkeyed, Awaiting Connection
0 1 0 0	Non–Volatile Configuration Bad
0 1 0 1	Major Fault (see bits 10 and 11 for fault classification and LED states) Minor fault is not a state change
0 1 1 0	Connected, Active
0 1 1 1	Idle (program mode type)
1 0 0 0 1 0 0 1	reserved, shall not be used
1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1	reserved for product specific states

4.1.8.2.2 Common services

4.1.8.2.2.1 Get_Attribute_All Response

At the **class level**, the order of the attributes returned in the “Attribute Data” parameter of the Get_Attribute_All response shall be as defined in Table 74.

Table 74 – Class level object/service specific response data of Get_Attribute_All

Attribute ID	Octet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	Revision (low octet) Default = 1							
	1	Revision (high octet) Default = 0							
2	2	Max Instance (low octet) Default = 1							
	3	Max Instance (high octet) Default = 0							
6	4	Max ID Number of Class Attributes (low octet) Default = 0							
	5	Max ID Number of Class Attributes (high octet) Default = 0							
7	6	Max ID Number of Instance Attributes (low octet) Default = 0							
	7	Max ID Number of Instance Attributes (high octet) Default = 0							

At the **instance level**, the order of the attributes returned in the "Attribute Data" parameter of the Get_Attribute_All response shall be as defined in Table 75.

Table 75 – Instance level object/service specific response data of Get_Attribute_All

Attribute ID	Octet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	Vendor (low octet)							
	1	Vendor (high octet)							
2	2	Device Type (low octet)							
	3	Device Type (high octet)							
3	4	Product Code (low octet)							
	5	Product Code (high octet)							
4	6	Major Revision							
	7	Minor Revision							
5	8	Status (low octet)							
	9	Status (high octet)							
6	10	Serial Number (low octet)							
	11	Serial Number							
	12	Serial Number							
	13	Serial Number (high octet)							
7	14	Product Name length							
	15	Product Name (1st character)							
	16	Product Name (2nd character)							
	n	Product Name (last character)							

Because the length of the name is not known before issuing the Get_Attribute_All service request, implementors shall allow enough memory space to store a response up to 32 characters in length.

To maintain consistency, if any instance attributes are added at some later date, the instance Get_Attribute_All response shall be modified to include the attributes as defined in Table 76.

Table 76 – Modified instance level object/service specific response data of Get_Attribute_All

Attribute ID	Octet	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
8	n+1	State Default = 255							
9	n+2	Space reserved for attribute 9 low octet, default = 0							
	n+3	Space reserved for attribute 9 high octet, default = 0							
10	n+4	Heartbeat Interval, Default = 0							

4.1.8.2.2.2 Reset service

The Reset common service shall have the object–specific parameter specified in Table 77.

Table 77 – Object-specific parameter for Reset

Name	Type	Semantics of values
Type	USINT	Value 0 shall emulate as closely as possible cycling power on the item the Identity object represents. This value shall be the default if this parameter is omitted. Value 1 shall return as closely as possible to the out–of–box configuration, then shall emulate cycling power as closely as possible. Value 2 shall return as closely as possible to the out-of-box configuration with the exception of communication link parameters, then emulate cycling power as closely as possible. The communication link parameters that are to be preserved are defined by each network type. Values 3-255 are reserved.

4.1.8.3 Message Router object specific syntax elements

4.1.8.3.1 Attributes

4.1.8.3.1.1 Class attributes

The Message Router object class attributes shall be as specified in Table 78.

Table 78 – Message Router object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 1
4	Optional attribute list	STRUCT of	
	Number of attributes	UINT	
	Optional attributes	ARRAY of UINT	
5	Optional service list	STRUCT of	
	Number services	UINT	
	Optional services	ARRAY of UINT	
6	Max ID Number of Class Attributes	UINT	
7	Max ID Number of Instance Attributes	UINT	

4.1.8.3.1.2 Instance attributes

The format of the Message Router object instance attributes is as specified in Table 79.

Table 79 – Message Router object instance attributes

Attribute ID	Name	Data type
1	Object_List	STRUCT of
	Number	UINT
	Classes	ARRAY of UINT
2	Number available	UINT

4.1.8.3.2 Common services

Get_Attribute_All Response

At the **class level**, the order of the attributes returned in the "Attribute Data" parameter of the Get_Attribute_All response shall be as specified in Table 80. Default values for all unsupported class attributes shall be as shown in that table.

Table 80 – Class level object/service specific response data of Get_Attribute_All

Class attribute ID	Attribute name, description and default value
1	Revision default = 1
4	Optional attribute list, number of attributes default = 0
5	Optional service list, number of services default = 0
6	Max ID number of class attributes default = 0
7	Max ID number of instance attributes default = 0

Default values for all unsupported instance attributes shall be as shown in Table 81.

Table 81 – Instance level object/service specific response data of Get_Attribute_All

Instance attribute ID	Attribute name, description and default value
1	Object list number, number of supported classes default = 0
2	Number available, maximum number of connections default = 0
3	Always = 0x0000

4.1.8.4 Assembly object specific syntax elements

4.1.8.4.1 Attributes

4.1.8.4.1.1 Class attributes

The format of the Assembly object class attributes shall be as specified in Table 82.

Table 82 – Assembly object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 2
2	Max Instance	UINT	

4.1.8.4.1.2 Instance attributes

The format of the Assembly object instance attributes is as specified in Table 83.

Table 83 – Assembly object instance attributes

Attribute ID	Name	Data type	Semantics of values
1	Number of Members in List	UINT	
2	Member List	ARRAY of STRUCT	This attribute has a complex data type
	Member Data Size	UINT	Size in bits
	Member Path Size	UINT	Size in octets (0 = Empty Path)
	Member Path	Packed EPATH	Packed path to the member data. See 4.1.8.8 for definition of Path
3	Data	ARRAY of SWORD	Contain all of the member data packed into one array. This data may contain many different data types. For efficiency it is best to keep this data word aligned by packing it on word boundaries and adding padding as needed. This can be accomplished by using "empty paths" (Member Path Size = 0)
4	Size	UINT	

4.1.8.5 Acknowledge Handler object specific syntax elements

4.1.8.5.1 Attributes

4.1.8.5.1.1 Class attributes

The format of the Acknowledge Handler object class attributes shall be as specified in Table 84.

Table 84 – Acknowledge Handler object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 1
4	Optional attribute list	STRUCT of	
	Number of attributes	UINT	
	Optional attributes	ARRAY of UINT	
5	Optional service list	STRUCT of	
	Number services	UINT	
	Optional services	ARRAY of UINT	
6	Max ID Number of Class Attributes	UINT	
7	Max ID Number of Instance Attributes	UINT	

4.1.8.5.1.2 Instance attributes

The format of the Acknowledge Handler object instance attributes is as specified in Table 85.

Table 85 – Acknowledge Handler object instance attributes

Attribute ID	Name	Data type	Semantics of values
1	Acknowledge Timer	UINT	Range 1 – 65 535 ms (0 invalid) Default = 16
2	Retry Limit	USINT	Range 0 – 255 Default = 1
3	COS Producing Connection Instance	UINT	Connection Instance ID
4	Ack List Size	SWORD	0 = Dynamic > 0 = Max number of members
5	Ack List	STRUCT of	
		SWORD	Number of members in the array
		ARRAY of UINT	List of Connection Instance IDs
6	Data with Ack Path List Size	SWORD	0 = Dynamic > 0 = Max number of members
7	Data with Ack Path List	STRUCT of	
		SWORD	Number of members in the array
		ARRAY of	
		UINT	Connection Instance ID
		USINT	Path length
		Padded EPATH	Path

4.1.8.5.2 Object specific services

4.1.8.5.2.1 Add_AckData_Path

The Add_AckData_Path_RequestPDU body shall be as specified in Table 86.

Table 86 – Structure of Add_AckData_Path_RequestPDU body

Name	Data type
Connection Instance ID	UINT
Consumer Path Size	USINT
Consumer Path	Padded EPATH

The Add_AckData_Path_ResponsePDU body is empty.

4.1.8.5.2.2 Remove_AckData_Path

The Add_AckData_Path_RequestPDU body shall be as specified in Table 87.

Table 87 – Structure of Remove_AckData_Path_RequestPDU body

Name	Data type
Connection Instance ID	UINT

The Remove_AckData_Path_ResponsePDU body is empty.

4.1.8.6 Time Sync object specific syntax elements

4.1.8.6.1 Attributes

Instance attributes

The format of the Time Sync object instance attributes shall be as specified in Table 88.

Table 88 – Time Sync object instance attributes

Attribute ID	Name	Data Type	Semantics of values
1	EnablePTP	BOOL	PTP enabled for this device: 0 = PTP not enabled 1 = PTP enabled
2	IsSynchronized	BOOL	Local clock synchronized with reference clock: 0 = local clock not synchronized 1 = local clock synchronized
3	CurrentTimeMicroseconds	LINT	Current value of local_time in microseconds
4	CurrentTimeNanoseconds	LINT	Current value of local_time in nanoseconds
5	OffsetToMaster	LINT	Offset between local clock and master clock in nanoseconds
6	MaxOffsetToMaster	LINT	Maximum offset between local clock and master clock in nanoseconds
7	DelayToMaster	LINT	Path delay to master
8	GrandMasterClockInfo	STRUCT of	Grandmaster Clock Info
	Identifier	STRING[4]	Clock Identifier
	Stratum	INT	Clock Stratum (0 – 255)
	Variance	INT	Clock Variance
	CommunicationTechnology	INT	Clock communication technology: CPF 2/1 = 9 CPF 2/2 = 1 CPF 2/3 = 7
	PortId	INT	Port Identifier
	UUID	ARRAY of SINT[6]	Clock UUID CPF 2/1 Vendor Id / Serial Number CPF 2/2 MAC Address CPF 2/3 Vendor Id / Serial Number
9	ParentClockInfo	STRUCT of	Parent Clock Info
	Reserved	DINT	reserved
	ObservedDrift	DINT	Observed Drift
	ObservedVariance	INT	Observed Variance
	Variance	INT	Clock Variance
	CommunicationTechnology	INT	Clock communication technology: CPF 2/1 = 9 CPF 2/2 = 1 CPF 2/3 = 7
	PortId	INT	Port Identifier
	UUID	ARRAY of SINT[6]	Clock UUID CPF 2/1 Vendor Id / Serial Number CPF 2/2 MAC Address CPF 2/3 Vendor Id / Serial Number

Attribute ID	Name	Data Type	Semantics of values
10	LocalClockInfo	STRUCT of	Local Clock Info
	Identifier	STRING[4]	Clock Identifier
	Stratum	INT	Clock Stratum
	Variance	INT	Clock Variance
	CommunicationTechnology	INT	Clock communication technology: CPF 2/1 = 9 CPF 2/2 = 1 CPF 2/3 = 7
	PortId	INT	Port Identifier
	UUID	ARRAY of SINT[6]	Clock UUID CPF 2/1 Vendor Id / Serial Number CPF 2/2 MAC Address CPF 2/3 Vendor Id / Serial Number
11	NumberOfPorts	INT	Number of ports
12	PortState	STRUCT of	Port states
		INT	Number of ports
		ARRAY of SINT	Port state of each port
13	PortEnable	STRUCT of	Port enable status
		INT	Number of ports
		ARRAY of SINT	Port enable status of each port: 0 = port disabled 1 = port enabled
14	PortBurstEnable	STRUCT of	Port burst enable status
		INT	Number of ports
		ARRAY of SINT	Port burst enable status of each port: 0 = port burst disabled 1 = port burst enabled
15	SyncInterval	SINT	Logarithm base 2 of the sync interval Default = 2 seconds
16	PreferredMaster	BOOL	Preferred Master: 0 = clock is not preferred 1 = clock is preferred Default = 0
17	Subdomain	STRING[16]	PTP Clock subdomain Default = "_DFLT"
18	ClockMode	SINT	Clock Mode: Slave Only / Ordinary Clock = 0 Master Capable / Ordinary Clock = 1 Master Capable / Boundary Clock = 2
19	StepsRemoved	INT	Number of communication paths between the local and the grandmaster clocks
20	SystemTimeOffset	LINT	Offset to the local clock value
NOTE See IEC 61588:2004 for more details.			

4.1.8.6.2 Object specific services

4.1.8.6.2.1 Initialize

The Initialize_RequestPDU body and Initialize_ResponsePDU body shall be as specified in IEC 61588:2004.

4.1.8.6.2.2 Management_Message service

The Management_Message_RequestPDU body shall be as specified in Table 89.

Table 89 – Structure of Management_Message_RequestPDU body

Name	Data Type	Semantics of values
Management Message Command	UINT	See Table 91
Management Message Parameters	Command specific	See Management Message Key sections in IEC 61588:2004.

The Management_Message_ResponsePDU body shall be as specified in Table 90.

Table 90 – Structure of Management_Message_ResponsePDU body

Name	Data Type	Semantics of values
Management Message Command	UINT	Identical to request parameter. See Table 91
Management Message Parameters	Command specific	See Management Message Key sections in IEC 61588:2004.

Table 91 shows command values for the Management Message command

Table 91 – Management Message Command values

Management Message Command	Value
ObtainIdentity	1
InitializeClock	3
SetSubDomain	4
ClearDesignatedPreferredMaster	5
SetDesignatedPreferredMaster	6
GetDefaultDataSet	7
UpdateDefaultDataSet	9
GetCurrentDataSet	10
GetParentDataSet	12
GetPortDataSet	14
GetGlobalTimeDataSet	16
UpdateGlobalTimeProperties	18
GoToFaultyState	19
GetForeignDataSet	20
SetSyncInterval	22
DisablePort	23
EnablePort	24
DisableBurst	25
EnableBurst	26
SetTime	27
NOTE These values are specified in IEC 61588:2004.	

4.1.8.7 Connection Manager object specific syntax elements

4.1.8.7.1 Attributes

4.1.8.7.1.1 Class attributes

The format of the Connection Manager object class attributes shall be as specified in Table 92.

Table 92 – Connection Manager object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 1
2	Max Instance	UDINT	
4	Optional Attribute List	STRUCT of	
	Number attributes	UINT	
	Optional attributes	ARRAY of UINT	

4.1.8.7.1.2 Instance attributes

The format of the Connection Manager object instance attributes is as specified in Table 93

Table 93 – Connection Manager object instance attributes

Attribute ID	Name	Data type	Semantics of values
1	OpenReqs	UINT	
2	OpenFormat Rejects	UINT	
3	OpenResource Rejects	UINT	
4	OpenOther Rejects	UINT	
5	CloseReqs	UINT	
6	CloseFormat Rejects	UINT	
7	CloseOther Rejects	UINT	
8	ConnTimeouts	UINT	Total number of connection timeouts that have occurred in connections controlled by this Connection Manager.
9	Connection Entry List	STRUCT of	
	NumConnEntries	UINT	Number of bits used in the ConnOpenBits element. This attribute, divided by 8 and incremented for any remainder, gives the length in octets of the array in the ConnOpenBits field.
	ConnOpenBits	ARRAY of BOOL	Bit field. List of connection data. Each bit represents a possible connection. 0 = No Connection. 1 = Connection Established.
11	CpuUtilization	UINT	CPU Utilization in tenths of a percent. Range of 0 – 1 000 representing 0 to 100%.
12	MaxBuffSize	UDINT	Amount of buffer space is in octets.
13	BufSize Remaining	UDINT	Amount of buffer space is in octets.

4.1.8.8 Connection object specific syntax elements

4.1.8.8.1 Attributes

4.1.8.8.1.1 Class attributes

The format of the Connection object class attributes shall be as specified in Table 94.

Table 94 – Connection object class attributes

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	Revision of this object. Value = 1
2	Max Instance	UINT	

4.1.8.8.1.2 Instance attributes

The format of the Connection object instance attributes is as specified in Table 95.

Table 95 – Connection object instance attributes

Attr ID	Name	Data Type	Semantics
1	State	USINT	State of the object.
2	Instance_type	USINT	Indicates either I/O or Messaging Connection.
3	TransportClass_trigger	SWORD	Defines behavior of the Connection.
4	CP2/3_produced_ connection_id	UINT	Placed in ISO 11898 CAN Identifier Field when the Connection transmits on a CP 2/3 subnet.
5	CP2/3_consumed_ connection_id	UINT	ISO 11898CAN Identifier Field value that denotes message to be received on a CP 2/3 subnet.
6	CP2/3_initial_comm_ characteristics	SWORD	Defines the Message Group(s) across which productions and consumptions associated with this Connection occur on a CP 2/3 subnet.
7	Produced_connection_size	UINT	Maximum number of octets transmitted across this Connection.
8	Consumed_connection_size	UINT	Maximum number of octets received across this Connection.
9	Expected_packet_rate	UINT	Defines timing associated with this Connection.
10	CFP2_produced_connection _id	UDINT	Identifies the message sent on the subnet by this connection.
11	CPF2_consumed_connection _id	UDINT	Identifies the message received from the subnet for this connection.
12	Watchdog_timeout_action	USINT	Defines how to handle Inactivity/Watchdog timeouts.
13	Produced_connection_path_ length	UINT	Number of octets in the produced_connection_path attribute.
14	Produced_connection_path	Packed EPATH	Specifies the application object(s) whose data is to be produced by this Connection object. See 4.1.9.
15	Consumed_connection_path _ length	UINT	Number of octets in the consumed_connection_path attribute.
16	Consumed_connection_path	Packed EPATH	Specifies the application object(s) that are to receive the data consumed by this Connection object. See 4.1.9.
17	Production_inhibit_time	UINT	Defines minimum time between new data production. This attribute is required for all I/O Client connections, except those with a production trigger of Cyclic.
18	Connection_timeout_multipli er	USINT	Specifies the multiplier applied to the expected_packet_rate value to derive the value for the Inactivity/Watchdog Timer.
19	Connection_binding_list	STRUCT UINT Array of UINT	List of I/O connection instances bound to this instance.

State

This attribute defines the current state of the Connection instance. Table 96 defines the possible states and assigns a value used to indicate that state.

Table 96 – Values assigned to the state attribute

Value	State Name	Description
00	Non-existent	The Connection has yet to be instantiated.
01	Configuring	The Connection has been instantiated and is waiting for the following events to occur: (1) to be properly configured and (2) to be told to apply the configuration.
02	Waiting For Connection ID[a]	The Connection instance is waiting exclusively for its consumed_connection_id and/or produced_connection_id attribute to be set.[b]
03	Established	The Connection has been validly/fully configured and the configuration has been successfully applied.
04	Timed Out	If a Connection object experiences an Inactivity/Watchdog timeout, then a transition may be made to this state. See the watchdog_timeout_action attribute description and the description of the Inactivity/Watchdog for more details.
05	Deferred Delete[a]	If an Explicit Messaging Connection object experiences an Inactivity/Watchdog timeout, then a transition may be made to this state. See the watchdog_timeout_action attribute description and the description of the Inactivity/Watchdog Timer for more details.
06	Closing	A Bridged Connection object has received, and is processing, a Forward Close from the Connection Manager. The deletion of the connection does not occur until after a successful Forward Open response has been received from the target node.

[a] This value is only used on CP 2/3.

[b] When the Connection instance attributes are applied it may not be possible for the module to generate the produced_connection_id and/or consumed_connection_id values (see initial_comm_characteristics attribute for rules). If this is the case then all the tasks required to apply the attributes are performed except for the initialization of these attributes within the Connection and associated Link Producer/Consumer objects and a transition is made to this state. In this state the Connection instance is waiting exclusively for its produced_connection_id and/or consumed_connection_id attributes to be set.

Important: A dynamically created connection instance is the child of the Explicit Messaging connection across which it was created. Different subnet types may provide other mechanisms for creating a connection that imply a particular parent-child relationship. See network-specific specifications for details.

Important: All resources associated with a Connection Instance (*A)* that has been dynamically created across an Explicit Messaging Connection (*B*) shall be released if the Explicit Messaging Connection (*B)* times out prior to the dynamically created Connection Instance (*A)* transitioning to the Established state.

Important: When a transition is made to the Established state, all timers associated with the Connection object are activated (see Connection Timing in IEC 61158-5-2, 6.2.3.2.1.7)

Instance_type

This attribute defines the instance type (see Table 97).

Table 97 – Values assigned to the instance_type attribute

Value	Meaning
00	Explicit Messaging. This Connection Instance represents one of the end-points of an Explicit Messaging Connection. An Explicit Messaging Connection is dynamically created by sending the *Open Explicit Messaging Connection Request* to the Connection Class.
01	I/O. This Connection Instance represents one of the end-points of an I/O Connection. An I/O Connection is dynamically created by sending a *Create Request* to the Connection Class.
02	Bridged. This Connection Instance represents an intermediate 'hop' of a bridged I/O or Explicit Messaging connection. A pair of Bridged Connection objects (one for each subnet bridged between) are dynamically created by a node after successfully receiving a Forward Open service to the Connection Manager object when this node is not the end point (target).

TransportClass_trigger

Defines whether this is a producing only, consuming only, or both producing and consuming connection. If this end point is to perform a data production, this attribute also defines the event that triggers the production. The eight (8) bits are divided as shown in Figure 7.

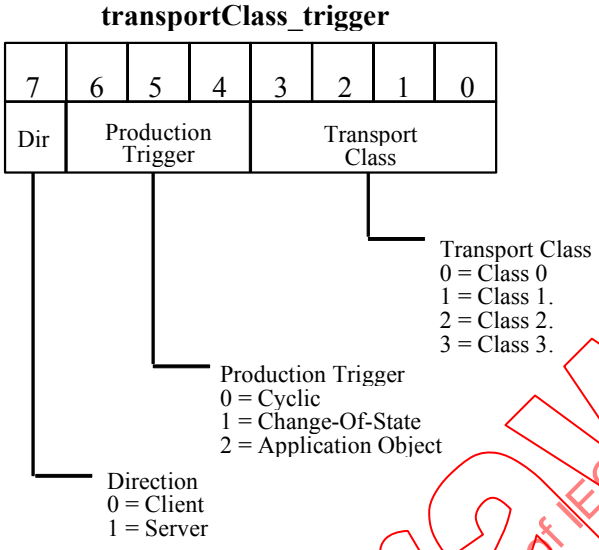


Figure 7 – Transport Class Trigger attribute

The ***Direction bit*** of the transportClass_trigger octet indicates whether the end-point is to act as the Client or the Server on this connection (see Table 98).

Table 98 – Possible values within Direction Bit

Value	Meaning	
0	Client	This end-point provides the Client behavior associated with this Connection. Additionally, this value indicates that the *Production Trigger* bits within the transportClass_trigger octet contain the description of when the Client is to produce the message associated with this connection. Client connections with production trigger value of 0 or 1 (Cyclic or Change-of-State) shall produce immediately after transitioning to the Established state.
1	Server	This end-point provides the Server behavior associated with this Connection. In addition, this value indicates that the *Production Trigger* bits within the transportClass_trigger octet are to be IGNORED. The *Production Trigger* bits are ignored due to the fact that a Server end-point *reacts* to the transmission from the Client. The only means by which a Server end-point is triggered to transmit is when this *reaction* calls for the production of a message (Transport Classes 2 or 3).

Table 99 lists the values that are possible within the ***Production Trigger*** bits of the **transportClass_trigger** attribute.

Table 99 – Possible values within Production Trigger Bits

If the value is:	Then the Production of a message is:	
0	Cyclic	The expiration of the Transmission Trigger Timer triggers the data production. See IEC 61158-5-2, 6.2.3.2.1.7 Connection Timing for a detailed description of the Transmission Trigger Timer.

If the value is:	Then the Production of a message is:	
1	Change-Of-State	Production occurs when a change-of-state is detected by the application object. NOTE The consuming end-point may have been configured to expect the packet at a certain rate, regardless of the triggering mechanism at the producing end-point. See the description of the expected_packet_rate attribute of a Connection object and the description of Connection Timing in IEC 61158-5-2, 6.2.3.2.1.7 for more information.
2	Application Object Triggered	The application object decides when to trigger the production. NOTE The consuming end-point may have been configured to expect the packet at a certain rate, regardless of the triggering mechanism at the producing end-point. See the description of the expected_packet_rate attribute of a Connection object and the description of Connection Timing in IEC 61158-5-2, 6.2.3.2.1.7 for more information.
3 - 7	Reserved	

Table 100 lists possible values within the ***Transport Class*** nibble of the **transportClass_trigger** attribute. Behaviors resulting from these particular values are illustrated in the series of figures that follow the table.

Table 100 – Possible values within Transport Class Bits

Value	Meaning	
0	Transport Class 0	Based on the value within the *Dir* bit, this connection end-point will be a producing only OR consuming only end-point. Upon application of this Connection instance, the module instantiates either a Link Producer (*Dir* bit = Client, producing only) or a Link Consumer (*Dir* bit = Server, consuming only) to be associated with this Connection.
1	Transport Class 1	
2	Transport Class 2	Indicates that the module will both produce AND consume across this connection. The Client end-point generates the first data production that is consumed by the Server, which causes the Server to return a production that is consumed by the Client.
3	Transport Class 3	
4	Transport Class 4	Non-blocking.
5	Transport Class 5	Non-blocking, fragmenting.
6	Transport Class 6	Multicast, fragmenting.
7 - F	Reserved	

A 16-bit sequence count value is prepended to all Class 1, 2, and 3 transports. This value is used to detect delivery of duplicate data packets. Sequence count values are initialized on the first message production and incremented on each subsequent new data production. A resend of old data shall not cause the sequence count to change and a consumer shall ignore data when it is received with a duplicate sequence count. Consuming applications can use this mechanism to distinguish between new samples and old samples that were sent to maintain the connection.

However, on CP 2/3, transport classes 2 and 3 do not prepend a 16-bit sequence count as described in 4.1.4.

The following tables and figures illustrate the valid combinations of ***Production Trigger*** and ***Transport Class*** and provides a description of the Client and Server behaviors. See IEC 61158-5-2, 6.2.3.2.1.7 for a description of the *Transmission Trigger Timer,* which is shown in the illustrations.

To summarize, refer to Table 101 for the valid values within the **transportClass_trigger** attribute of a Connection Instance:

Table 101 – Transport Class_Trigger attribute

TransportClass_trigger bits	Meaning
1 xxx 0000	Direction = Server, Production Trigger = IGNORED, Transport Class = 0.
1 xxx 0001	Direction = Server, Production Trigger = IGNORED, Transport Class = 1.
1 xxx 0010	Direction = Server, Production Trigger = IGNORED, Transport Class = 2.
1 xxx 0011	Direction = Server, Production Trigger = IGNORED, Transport Class = 3. This is the value assigned to this attribute within the Server end-point of an Explicit Messaging Connection.
0 000 0000	Direction = Client, Production Trigger = Cyclic, Transport Class = 0.
0 000 0001	Direction = Client, Production Trigger = Cyclic, Transport Class = 1.
0 000 0010	Direction = Client, Production Trigger = Cyclic, Transport Class = 2.
0 000 0011	Direction = Client, Production Trigger = Cyclic, Transport Class = 3.
0 001 0000	Direction = Client, Production Trigger = Change-Of-State, Transport Class = 0.
0 001 0001	Direction = Client, Production Trigger = Change-Of-State, Transport Class = 1.
0 001 0010	Direction = Client, Production Trigger = Change-Of-State, Transport Class = 2.
0 001 0011	Direction = Client, Production Trigger = Change-Of-State, Transport Class = 3.
0 010 0000	Direction = Client, Production Trigger = Application object, Transport Class = 0.
0 010 00001	Direction = Client, Production Trigger = Application object, Transport Class = 1
0 010 0010	Direction = Client, Production Trigger = Application object, Transport Class = 2.
0 010 0011	Direction = Client, Production Trigger = Application object, Transport Class = 3. This is the value assigned to this attribute within the Client end-point of an Explicit Messaging Connection.
1 111 1111	Default value assigned to this attribute within an I/O Connection.

CP2/3_produced_connection_id

This attribute is Required for a CP 2/3 subnet. Other subnet types shall not use this attribute.

It contains the CP 2/3 Connection ID to be associated with transmissions sent across this connection (if any), i.e. the value that will be specified in the CAN Identifier Field of ISO 11898 when this Connection transmits. See IEC 62026-3 for a description of how CP 2/3 uses the CAN Identifier Field. This value is loaded directly into the associated Link Producer's connection_id attribute. Values are defined in Table 102.

Table 102 – Values defined for the CP2/3_produced_connection_id attribute

Value	Meaning
0 - $7F0_{hex}$	The value to be placed in the CAN Identifier Field when this Connection transmits.
800_{hex} - $FFFE_{hex}$	Reserved
$FFFF_{hex}$	Default value assigned to this attribute within an I/O Connection. This attribute will retain this value if this Connection instance is not producing any data (consumer only).

CP2/3_consumed_connection_id

This attribute is Required for a CP 2/3 subnet. Other subnet types shall not use this attribute.

It contains the Connection ID which identifies messages to be received across this connection (if any), i.e. the CAN Identifier Field value that is associated with messages this Connection object receives. See IEC 62026-3 for a description of how CP 2/3 uses the CAN Identifier Field. This value is loaded directly into the associated Link Consumer's connection_id attribute. Values are defined in Table 103.

Table 103 – Values defined for the CP2/3_consumed_connection_id attribute

Value	Meaning
0 - $7F0_{hex}$	The value that identifies messages to be consumed. This will be specified in the CAN Identifier Field of messages that are to be consumed.
800_{hex} - $FFFE_{hex}$	Reserved
$FFFF_{hex}$	Default value assigned to this attribute within an I/O Connection. This attribute will retain this value if this Connection instance is not consuming any data (producer only).

CP2/3_initial_comm_characteristics

This attribute is Required for a CP 2/3 subnet. Other subnet types shall not use this attribute.

It defines the Message Group(s) across which productions and consumptions associated with this Connection occur.

This octet is divided into two nibbles, as show in Figure 8.

7	6	5	4	3	2	1	0
Initial Production Characteristics				Initial Consumption Characteristics			

Figure 8 – CP2/3_initial_comm_characteristics attribute format

Table 104 lists the values that are possible within the Initial Production Characteristics nibble (upper nibble) of the CP2/3_initial_comm_characteristics attribute.

Table 104 – Values for the Initial Production Characteristics nibble

Value	Meaning	
0	Produce across Message Group 1	The production associated with this Connection is to take place across Message Group 1. The producing module generates the Connection ID value and loads it into the Connection object's CP2/3_produced_connection_id attribute. The producing module allocates a Message ID from its Group 1 Message ID pool and combines this with its Source MAC ID to generate the Connection ID. The numerically lowest available Group 1 Message ID is to be used in generating the CP2/3_produced_connection_id attribute value. This value shall also be loaded into the corresponding CP2/3_consumed_connection_id attribute(s) associated with the consuming Connection object(s).
1	Produce across Message Group 2 (Destination)	The production associated with this Connection is to take place across Message Group 2. Additionally, the intended recipient's MAC ID (Destination MAC ID) is to be placed within the MAC ID component of the Group 2 Identifier Field. In this case, the consuming module generates the Connection ID value to be associated with transmissions across this connection. When the consuming module has generated this value and loaded it into the appropriate Connection object's CP2/3_consumed_connection_id attribute, it can be read and subsequently loaded into the producing Connection object's CP2/3_produced_connection_id attribute.
2	Produce across Message Group 2 (Source)	The production associated with this Connection is to take place across Message Group 2. In addition, the producing module's MAC ID (Source MAC ID) is to be placed within the MAC ID component of the Group 2 Identifier. In this case, the producing module generates the Connection ID value and loads it into the Connection object's CP2/3_produced_connection_id attribute. The numerically lowest available Group 2 Message ID is to be used in generating the CP2/3_produced_connection_id attribute value. This value shall also be loaded into the corresponding CP2/3_consumed_connection_id attribute(s) associated with the consuming Connection object(s).
3	Produce across Message Group 3	The production associated with this Connection is to take place across Message Group 3. The producing module generates the Connection ID value and loads it into the Connection object's CP2/3_produced_connection_id attribute. The producing module allocates a Message ID from its Group 3 Message ID pool and combines this with its Source MAC ID to generate the Connection ID. The numerically lowest available Group 3 Message ID is to be used in generating the CP2/3_produced_connection_id attribute value. This value shall also be loaded into the corresponding CP2/3_consumed_connection_id attribute(s) associated with the consuming Connection object(s).
4 - 0xE	Reserved	
0xF	Default value	The default value assigned to the CP2/3 Initial Production Characteristics nibble within an I/O Connection. NOTE If this is a consuming only I/O Connection, then the default value remains in this nibble. Explicit Messaging Connection objects automatically configure this attribute when the Connection is established.

Table 105 lists the possible values within the Initial Consumption Characteristics nibble (lower nibble) of the CP2/3_initial_comm_characteristics attribute.

Table 105 – Values for the Initial Consumption Characteristics nibble

Value	Meaning	
0	Consume a Group 1 Message	The message to be consumed will be transmitted across Message Group 1. The producing module generates the Connection ID value. This value shall be loaded into the CP2/3_consumed_connection_id attribute associated with the consuming Connection object(s).
1	Consume a Group 2 Message (Destination)	The message to be consumed will be transmitted across Message Group 2. The intended recipient's MAC ID (Destination MAC ID) is specified within the Group 2 Identifier. The consuming module generates the Connection ID value and loads it into the CP2/3_consumed_connection_id attribute associated with this Connection object. The numerically lowest available Group 2 Message ID is to be used in generating the CP2/3_consumed_connection_id attribute value. This value shall be loaded into the producing Connection object's CP2/3_produced_connection_id attribute.
2	Consume a Group 2 Message (Source)	The message to be consumed will be transmitted across Message Group 2. The transmitting module's MAC ID (Source MAC ID) is specified within the Group 2 Identifier. In this case, the producing module generates the Connection ID value and loads it into the Connection object's the CP2/3_produced_connection_id attribute. This value shall be loaded into the CP2/3_consumed_connection_id attribute associated with the consuming Connection object(s).
3	Consume a Group 3 Message	The message to be consumed will be transmitted as a Group 3 Message. The producing module generates the Connection ID value. The Connection ID value shall be loaded into this Connection object's CP2/3_consumed_connection_id attribute.
4 - 0xE	Reserved	
0xF	Default value	The default value assigned to the CP 2/3 Initial Consumption Characteristics nibble within an I/O Connection. NOTE If this is a producing only I/O Connection, then the default value remains in this nibble. Explicit Messaging Connections automatically configure this attribute when the Connection is established.

Important: The module that generates a Connection ID shall guarantee that it does not allocate the Message ID/MAC ID pair in such a way that two separate modules are capable of transmitting identical bit patterns within the Identifier Field.

Produced_connection_size

The meaning of this attribute is different for Explicit Messaging Connections than it is for I/O Connections. If the subnet defines a fragmentation protocol and the device supports fragmentation, this size may be larger than the largest frame size. See the network specific adaptation specifications for more details.

For Explicit Messaging Connections:

This attribute signifies the maximum number of Message Router Request/Response Data octets (see 4.1.8) that a device is able to transmit across this Connection. Devices that place a known limit on the maximum amount of Message Router Request/Response Data that can be transmitted in a single message, or single fragmented series initialize this attribute accordingly. Devices that cannot or do not predefine an up-front transmit limit place the value 0xffff into this attribute (there may still be a limit, however, it is not known in advance).

Important: Due to the nature of Explicit Messaging, the length of Explicit Messages will fluctuate over the lifetime of a connection. Explicit Messaging Connections perform fragmentation based on the *length* of the current message to transmit where the subnet type may define a fragmentation protocol.

For I/O Connections:

If the **transportClass_trigger** indicates that this Connection instance is to produce, then this attribute defines the maximum amount of I/O data that may be produced as a *single unit* across this connection. The amount of I/O to be transmitted at any given point in time can be less than or equal to the **connection_size** attribute.

This attribute defaults to zero (0) within an I/O Connection. If the subnet type supports fragmentation and this attribute is set to a value greater than the largest payload in an I/O Connection, then the Connection will break up the data into multiple fragments. See the network specific adaptation specifications for more details.

Important: Fragmentation within I/O Connections is performed based on the value within this attribute, regardless of the current amount of data to transmit.

Important: I/O Messages that contain no Application I/O Data and were configured to contain data (via produced_connection_size being greater than zero (0)) are defined to indicate a *No Data* event for the receiving application object(s). The behavior of an application object upon detection of the *No Data* event is application object specific.

Consumed_connection_size

The meaning of this attribute is different for Explicit Messaging Connections than it is for I/O Connections. If the subnet defines a fragmentation protocol and the device supports fragmentation, this size may be larger than the largest frame size. See the network specific adaptation specifications for more details.

For Explicit Messaging Connections:

This attribute signifies the maximum number of Message Router Request/Response Data octets that a device is able to receive across this Connection.

Devices that place a known limit on the maximum amount of Message Router Request/Response Data that can be received in a single message, or single fragmented series initialize this attribute accordingly. Devices that cannot or do not predefine an up-front receive limit place the value 0xffff into this attribute (there may still be a limit, however, it is not known in advance). Because of the nature of Explicit Messaging, the length of Explicit Messages will fluctuate over the lifetime of a connection.

For I/O Connections:

If the **transportClass_trigger** attribute indicates that this Connection is to consume, then this attribute defines the maximum amount of data that may be received as a *single unit* across this connection. The actual amount of I/O data received at any given time can be less than or equal to the **connection_size** attribute.

This attribute defaults to zero (0) within an I/O Connection. If the subnet type supports fragmentation and this attribute is set to a value greater than the largest payload in an I/O Connection, then the Connection will process the fragmentation protocol. See the network specific adaptation specifications for more details.

The length of an I/O Message shall be less than or equal to this attribute for an I/O Connection object to receive it as a valid message. If an I/O Connection object receives a message whose length is greater than this attribute, then it immediately discards the message and discontinues any subsequent processing.

NOTE With respect to the Server end-point of a Transport Class 2 or 3 Connection, this *too much data* error condition results in no response being transmitted and the watchdog timer is not kicked.

Expected_packet_rate

This attribute is used to generate the values loaded into the *Transmission Trigger Timer* and the *Inactivity/Watchdog Timer*. See IEC 61158-5-2, 6.2.3.2.1.7 for a description of the Transmission Trigger and Inactivity/Watchdog timers.

The resolution of this attribute is in milliseconds. A request to configure this attribute may result in the specification of a time value that a product cannot meet. In addition to performing product specific range checking when a request to modify this attribute is received, the following steps are performed:

- If the specified value is not equal to an increment of the available clock resolution, then the value is rounded **up** to the next serviceable value. For example: a Set_Attribute_Single request is received specifying the value 5 for the **expected_packet_rate** attribute and the product provides a 10 millisecond resolution on timers. In this case the product would load the value 10 into the **expected_packet_rate** attribute.
- The value that is actually loaded into the **expected_packet_rate** attribute is reported in the Service Data Field of a Set_Attribute_Single response message associated with a request to modify this attribute.
- If the requested value is equal to an increment of the clock resolution, then the requested value is loaded into the expected_packet_rate and reported in the response. For example: if the value 100 is requested and the clock resolution is 10 milliseconds, then of a value of 100 is loaded.
- When a Connection object is in the **Established** state, any modifications to the **expected_packet_rate** attribute have immediate effect on the Inactivity/Watchdog Timer. The following steps are performed by a Connection object in the **Established** state when a request is received to modify the **expected_packet_rate** attribute:
  - the current Inactivity/Watchdog Timer is canceled,
  - a new Inactivity/Watchdog Timer is activated based on the new value in the expected_packet_rate attribute.

This attribute defaults to 2500 (2 500 ms) within Explicit Messaging Connections, and to zero (0) within an I/O Connection.

CPF2_produced_connection_id

Contains the Connection ID, which identifies messages to be sent across this connection (if any). This attribute shall not be implemented when the subnet type is CP 2/3.

CPF2_consumed_connection_id

Contains the Connection ID, which identifies messages to be received across this connection (if any). This attribute shall not be implemented when the subnet type is CP 2/3.

Watchdog_timeout_action

This attribute defines the action the Connection object should perform when the Inactivity/Watchdog Timer expires. Table 106 defines the specifics of this attribute.

Table 106 – Values for the watchdog_timeout_action

Value	Meaning
0	*Transition to Timed Out*. The Connection transitions to the Timed Out state and remains in this state until it is Reset or Deleted. The command to Reset or Delete could come from an internal source (e.g., an application object) or could come from the network (e.g., a configuration tool). This is the default value for this attribute with respect to I/O Connections. This value is invalid for Explicit Messaging Connections.
1	*Auto Delete*. The Connection Class automatically deletes the Connection if it experiences an Inactivity/Watchdog timeout. This is the default value for this attribute with respect to Explicit Messaging Connections.
2	*Auto Reset*. The Connection remains in the Established state and immediately restarts the Inactivity/Watchdog timer. This value is invalid for Explicit Messaging Connections.
3	*Deferred Delete*. The Connection transitions to the Deferred state if any child connection instances are in the Established state. If no child connection instances are in the Established state the connection is deleted. This value is only used on CP 2/3 and is invalid for I/O Messaging Connections.
4 - FF	Reserved

Produced_connection_path_length

Specifies the number of octets of information within the **produced_connection_path** attribute. This is automatically initialized when the **produced_connection_path** attribute is configured. This attribute defaults to the value zero (0).

Produced_connection_path

The **produced_connection_path** attribute is made up of a octet stream which defines the application object(s) whose *data* is to be produced by this Connection object. **The format of this octet stream is specified in 4.1.9.** This attribute defaults to being empty upon instantiation of the Connection. It remains empty within Explicit Messaging Connections and within Connection objects that do not produce.

Consumed_connection_path_length

Specifies the number of octets of information within the **consumed_connection_path** attribute. This is automatically initialized when the **consumed_connection_path** attribute is configured. This attribute defaults to the value zero (0).

Consumed_connection_path

The **consumed_connection_path** attribute is made up of an octet stream which defines the application object(s) that are to receive the *data* consumed by this Connection object. **The format of this octet stream is specified in 4.1.9.** This attribute defaults to being empty upon instantiation of the Connection. It remains empty within Explicit Messaging Connections and within Connection objects that do not consume.

Production_inhibit_time

This attribute is used to configure the minimum delay time between new data production. This is required for all I/O Client connections, except those with a production trigger of Cyclic. The Set_Attribute_Single service shall be supported when this attribute is implemented. A value of zero (the default value for this attribute) indicates no inhibit time.

The resolution of this attribute is in milliseconds. A request to configure this attribute may result in the specification of time value that a product cannot meet. In addition to performing product specific range checking when a request to modify this attribute is received, the following steps are performed:

- If the specified value is not equal to an increment of the available clock resolution, then the value is rounded **up** to the next serviceable value. For example: a Set_Attribute_Single request is received specifying the value 5 for the **production_inhibit_time** attribute and the product provides a 10 millisecond resolution on times. In this case the product would load the value 10 into the **production_inhibit_time** attribute.

- The value that is actually loaded into the **production_inhibit_time** attribute is reported in the Service Data Field of a Set_Attribute_Single response message associated with a request to modify this attribute.
- If the requested value is equal to an increment of the clock resolution, then the requested value is loaded into the **production_inhibit_time** and reported in the response. For example: the value 100 is requested and the clock resolution is 10 milliseconds.

The **production_inhibit_time** value is loaded in the Production Inhibit Timer each time new data production occurs.

When a Connection object is in the **Established** state, any modifications to the **production_inhibit_time** attribute have no effect on a currently running Production Inhibit Timer. The new production_inhibit_time value is loaded into the Production Inhibit Timer on the following new data production.

When the apply_attributes service is received, the **production_inhibit_time** shall be verified against the **expected_packet_rate** attribute. If the **expected_packet_rate** value is greater than zero, but less than the **production_inhibit_time** value, then an error shall be returned. In this case, where two attribute values conflict use the **production_inhibit_time attribute** ID as the additional error code returned in the error response.

Connection_timeout_multiplier

The Connection_timeout_multiplier specifies the multiplier applied to the expected packet rate to obtain the value used by the Inactivity/Watchdog Timer. See the Connection Timeout Multiplier parameter description within the Connection Manager object Specific Service Parameters for the enumerated values of this attribute. The default value for this attribute is zero (specifying a multiplier of 4).

Connection_binding_list

The Connection_binding_list attribute identifies connection instances that are bound to this connection. The attribute structure provides a 16 bit count of bound connections, followed by a list of the bound instances. This attribute shall be supported if the Connection_Bind service is supported.

4.1.8.8.2 Object specific services

4.1.8.8.2.1 Connection_Bind Service

The Connection_Bind object specific service binds two dynamically created I/O connection instances together for purposes of connection timeouts and deletions. A dynamically created I/O connection is the result of a Create service to the Connection class with the Instance_type attribute set to I/O (attribute value of 1). If one of these I/O connection instance times out or is deleted, the other instance shall exhibit the same behavior.

The shall be as specified in Table 107.

Table 107 – Structure of Connection_Bind_RequestPDU body

Parameter Name	Data Type	Parameter Description
Bound Instances	STRUCT of UINT UINT	Instance numbers of Connection objects to be bound.

The Connection_Bind_ResponsePDU body is empty.

The service response may return a status from the list of status codes in Table 108.

Table 108 – Object specific status for Connection_Bind service

General Status Code	Extended Status Code	Status Description
0x02	0x01	One or both of the connection instances is non-existent.
0x02	0x02	The connection class and/or instance is out of resources to bind instances.
0x0C	0x01	Both of the connection instances are existent, but at least one is not in the Established state.
0x20	0x01	Both connection instances are the same value.
0xD0	0x01	One or both of the connection instances is not a dynamically created I/O connection.
0xD0	0x02	One or both of the connection instances were created internally and the device is not allowing a binding to it.

4.1.8.8.2.2 Producing_Application_Lookup Service

The Producing_Application_Lookup object specific service provides a mechanism to find one or more connection instances in the Established state producing data from a given application object. If the requested producing application path is being produced by any connection in the Established state, the instance number of each connection producing from that path is returned.

The Producing_Application_Lookup_RequestPDU body shall be as specified in Table 109.

Table 109 – Structure of Producing_Application_Lookup_RequestPDU body

Parameter Name	Data Type	Parameter Description
Producing Application Path	EPATH	Connection path of producing application to be searched for within connections in the Established state. The path shall be a single Logical or Symbolic EPATH.

The Producing_Application_Lookup_ResponsePDU body shall be as specified in Table 110.

Table 110 – Structure of Producing_Application_Lookup_ResponsePDU body

Parameter Name	Data Type	Parameter Description
Instance Count	UINT	Number of Instances returned in the Connection Instance List parameter.
Connection Instance List	Struct of UINT	List of instance numbers of connection producing the data from the requested connection path within the node.

The service response may return a status from the list of status codes in Table 111.

Table 111 – Producing_Application_Lookup Service status codes

General Status Code	Extended Status Code	Status Description
0x02	0x01	The connection path was not found in any connection instance in the Established state.

4.1.8.8.2.3 SafetyOpen Service

See IEC 61784-3-2 for the definition of this service.

4.1.8.8.2.4 SafetyClose Service

See IEC 61784-3-2 for the definition of this service.

4.1.9 Message and connection paths

4.1.9.1 Contents

The path shall specify the object element that is either the target of a connection request, or the destination of a message request. The path shall contain multiple segments which can indicate the route to the next node (in the case of multiple links), what to connect to or where to send a message in the target device. Each segment shall be comprised of a segment descriptor octet which specifies the segment type and segment information whose size is dependent on the segment type. The two types of path shall be

- padded paths;
- packed paths.

These two path formats (padded and packed) shall not be interchangeable. Usage of padded versus packed shall be dependent on the context of use. If no format is explicitly specified for a path, then padded format shall be used.

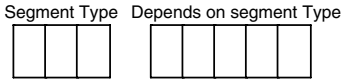
Each segment of a padded path shall be 16 bit word aligned. If a pad octet is required to achieve the alignment, then the segment shall specify the location of the pad octet. A packed path shall not contain the pad octets.

The possible segment types shall be as follows:

Port segment (where);
Logical segment (what);
Network segment (when or how);
Symbolic segment;
Data segment.

4.1.9.2 Segment type

Each segment of the path shall contain a segment type to indicate how the segment is to be interpreted. The segment type is contained in the first octet of the segment. The bits of the first octet of a path segment shall be numbered 0 to 7, where bit 0 shall be the least significant bit of the octet. Bits of the segment type are defined in Figure 9.



0	0	0	Port
0	0	1	Logical
0	1	0	Network
0	1	1	Symbolic
1	0	0	Data / Extended symbolic
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Reserved

Figure 9 – Segment type

4.1.9.3 Port segment

The port segment shall indicate

- communication port through which to leave the node (expressed as a 16-bit number);
- link address of the next device in the path.

Figure 10 shows the overall structure of the port segment.

The bits 5-7 of the first octet shall be zero to indicate the port segment type. Bit 4, the node/slot address size bit, shall be set to 0 if the link address is one octet. Bit 4 shall be set to 1 if the link address is larger than one octet. If the link address is larger than one octet, its size in octets shall be in the second octet of the port segment.

Bits 0 – 3, the port identifier, shall indicate through which port to leave the node. The port identifier shall specify a port number or an escape to an extended port identifier when the module can support more than 14 ports. Port number 0 shall be reserved. Port number 1 shall only be used to represent a back-plane port. If the port identifier is 15, then a 16-bit field, called the extended port identifier, shall be the next part of the port segment; otherwise, the value of the port identifier shall be the port number.

The port number shall be followed by a link address whose format depends on the port type. If the link address is greater than one octet, then it shall be padded so that the entire port segment is an even number of octets. The octet specifying the size of the link address shall not include the pad octet. The pad octet shall be set to 0.

NOTE 1 For the common port types, the link address is a single octet. Other port types, such as TCP/IP encapsulation, may require a larger link address (see 4.3 and 11).

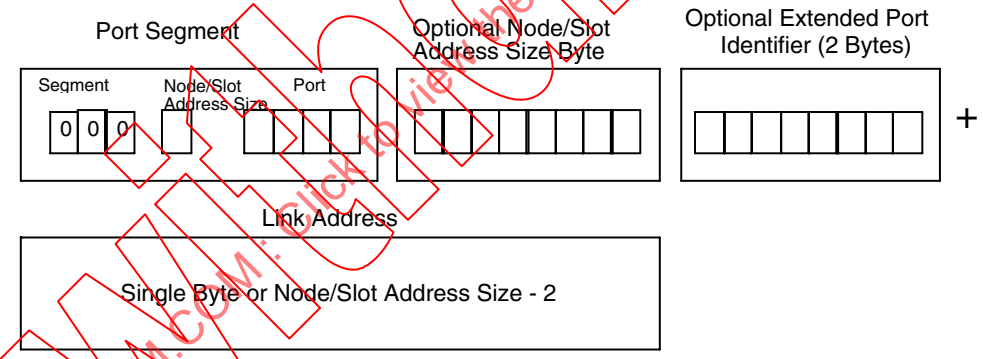


Figure 10 – Port segment

The Extended Port Identifier format of the Port Segment shall only be used when there are more than 14 ports possible on the device. The Port Segment shall always be packed into the smallest Port Segment format possible. Examples of possible port segments are as shown in Table 112.

Table 112 – Possible port segment examples

Port segment	Notes
[02][06]	Port 2, MAC ID 6
[15][0E][31][33][31][2E][31][35][31][2E][31][33][32][2E][35][35]	Multi-Octet Address for Ethernet Port 5, Link Address 131.151.132.55 (IP Address). The Address is defined as a character array.
[0F][12][00][01]	Port 18, MAC ID 1

The link address portion of a TCP/IP connection path segment shall be encoded within the port segment as a string of ASCII characters. The string shall be one of the following forms:

- IP address in dot notation, for example “130.151.132.55” (see RFC 1117 for the format of IP addresses);
- IP address in dot notation, followed by a “:” separator, followed by the TCP port number to be used at the specified IP address;
- Host name, for example “plc.type2.org”. The host name shall be resolved via a DNS request to a name server (see RFC 1035 for information on host names and name resolution);
- Host name, followed by a “:” separator, followed by the TCP port number to be used at the specified host.

The port number shall be represented in either hex or decimal. Hex shall be indicated by a leading “0x”. When a port number is specified, it shall be used rather than the standard port number used for the encapsulation protocol (0xAF12). Only port 0xAF12 is guaranteed to be available in a CP 2/2 compliant device.

NOTE 2 Other TCP port numbers may be implemented; however, this specification does not provide a mechanism to determine which TCP port numbers are supported by a device. The use of other TCP port numbers is therefore discouraged. The guaranteed TCP port number, 0xAF12, has been reserved with the Internet Assigned Numbers Authority (IANA) for use by the encapsulation protocol.

Since port segments shall be word-aligned, a pad octet may be required at the end of the string. The pad octet shall be 0x00, and shall not be counted in the Optional Address Size field of the port segment.

NOTE 3 Examples of port segments for TCP/IP are shown in Table 113 below.

Table 113 – TCP/IP link address examples

Port segment	IP address	Notes
[12][0D] [31][33][30][2E] [31][35][31][2E] [31][33][32][2E][31][00]	130.151.132.1	Multi-octet address for port 2, 13 octet string plus a pad octet
[13][12] [70][6C][63][2E] [63][6F][6E][74][72][6F][6C][6E][65][74][2E] [6F][72][67]	plc. type2.org	Multi-octet address for port 3, 18 octet string, no pad octet
[16][15] [31][33][30][2E] [31][35][31][2E] [31][33][32][2E] [35][35][3A] [30][78][33][32][31][30][00]	130.151.132.55:0x3210	Multi-octet address for port 6, 21 octet string plus a pad octet
[15][17] [70][6C][63][2E] [63][6F][6E][74][72][6F][6C][6E][65][74][2E] [6F][72][67][3A] [39][38][37][36][00]	plc. type2.org:9876	Multi-octet address port 5, 32 octet string plus a pad octet

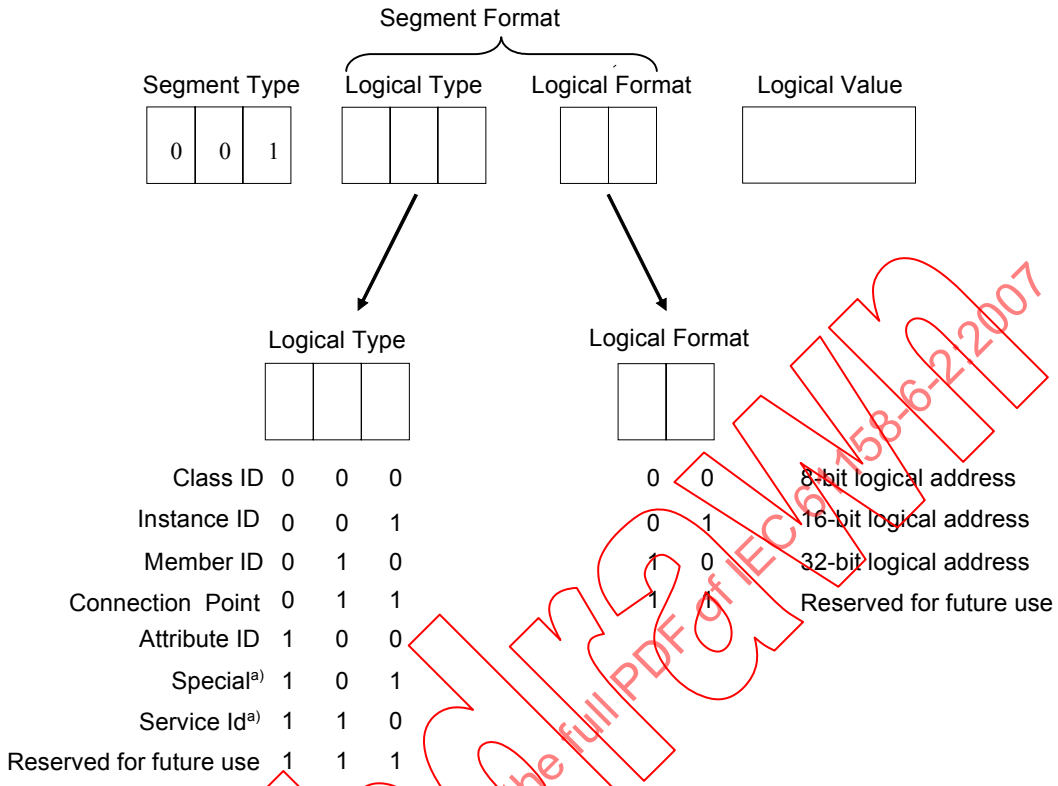
4.1.9.4 Logical segment

4.1.9.4.1 Logical segment structure

The logical segment selects a particular addressable entity within a device (for example, object class, object instance, and object attribute). When the logical segment is included within a Packed Path, the Logical Value shall be appended to the segment type octet with no pad in between. When the logical segment is included within a Padded Path, the 16-bit and 32-bit logical formats shall have a pad inserted between the segment type octet and the Logical Value (the 8-bit format is identical to the Packed Path). The pad octet shall be set to zero.

[a]The Special and Service ID Logical Types do not use the logical addressing definition for the Logical Format.

Figure 11 specifies the encoding of a logical segment.



[a]The Special and Service ID Logical Types do not use the logical addressing definition for the Logical Format.

Figure 11 – Logical segment encoding

The 8-bit and 16-bit logical address formats are allowed for use with all Logical Types.

The 32-bit logical address format is not allowed (reserved for future use).

The Class ID specifies a type of object within a device. The Instance ID is an integer assigned to an object when it is created. Instance 0, called the class instance, specifies the class itself. The attributes of an object are an enumerated list of externally visible values. The attribute ID specifies which attribute of an object is selected. Complex attributes are composed of members. The member ID specifies which member of the attribute.

The Connection Point Logical Type provides additional addressing capabilities beyond the standard Class ID/Instance ID/Attribute ID/Member ID object address. Object classes shall define when and how this addressing component is utilized (for example, a sub-instance of an Assembly object).

NOTE IEC 61158-5-2 describes the object model including the definition of class, instance, attribute, connection point, and members of a library of common objects.

The Service ID Logical Type has the following definition for the Logical Format:

- 0 0 8-Bit Service ID Segment (0x38)
- 0 1 Reserved for future use (0x39)
- 1 0 Reserved for future use (0x3A)
- 1 1 Reserved for future use (0x3B)

The Special Logical Type has the following definition for the Logical Format:

- 0 0 Electronic Key Segment (0x34)
- 0 1 Reserved for future use (0x35)
- 1 0 Reserved for future use (0x36)
- 1 1 Reserved for future use (0x37)

4.1.9.4.2 Electronic Key Segment

The electronic key segment shall be used to verify the type of target or router device. The key may be included as the first logical segment in a connection path and shall be checked by the Message Router of the receiving node against the values contained in instance 1 of its Identity object before any additional address checks are made. On a multi-hop message, sent to a remote link via intermediate routers, multiple key segments may appear. The format of the electronic key segment shall be as shown in Table 114.

Table 114 – Electronic key segment format

Parameter	Format	Value
segment_type	USINT	always 0x34
electronic_key_type	USINT	always 0x04
vendor_ID	UINT	these correspond to the values in instance 1 of the identity object
product_type	UINT	
product_code	UINT	
major_revision : 7	USINT	
compatible_match : 1	USINT	
minor_revision	USINT	

The `segment_type` for a key segment shall be 0x34. The `electronic_key_type` shall determine the format of a specific key segment and shall be set to 0x04. All other values for the `electronic_key_type` shall be reserved.

The `vendor_ID` shall specify the device vendor, or zero if no specific vendor is required.

The `product_type` shall specify a class of products such as digital input or analogue outputs. The `product_type` shall be ignored when it is set to zero. The `product_code` shall be vendor specific with each vendor having a unique code. The `product_code` shall be ignored when set to zero.

The `major_revision` and `compatible_match` fields shall be contained in the least significant 7 bits and most significant bit of the same octet, respectively.

The `major_revision` shall specify the functionality level of a device. The `major_revision` shall be ignored when set to zero.

The `compatible_match` shall modify the key matching function. If the bit is clear (= 0), then any non-zero `vendor_ID`, `product_type`, `product_code`, `major_revision`, and `minor_revision` shall match exactly. If the bit is set (= 1), the device may accept the key for any device which it can emulate. The level of emulation shall be product specific.

The `minor_revision` shall specify the revision of a device that does not effect network-visible behaviour. A value of zero shall specify that the originator of the request accepts any minor revision.

NOTE 2 IEC 61158-5-2 describes the Identity object and the rules for updating the revision fields.

4.1.9.4.3 Logical segments examples

Table 115.shows examples of logical segments.

Table 115 – Logical segments examples

First octet	Logical segment name	Padded format examples (as transmitted)	Packed format examples (as transmitted)	Notes
0x20	8-bit class	[20][04]	[20][04]	class 0x04 (Assembly object)
0x21	16-bit class	[21][00][34][12]	[21][34][12]	class 0x1234
0x24	8-bit instance	[24][04]	[24][04]	instance 0x04
0x25	16-bit instance	[25][00][34][12]	[25][34][12]	instance 0x1234
0x28	8-bit member	[28][04]	[28][04]	member 0x04
0x29	16-bit member	[29][00][34][12]	[29][34][12]	member 0x1234
0x2A	32-bit member			
0x2C	8-bit connection point	[2C][04]	[2C][04]	connection point 0x04
0x2D	16-bit connection point	[2D][00][34][12]	[2D][34][12]	connection point 0x1234
0x30	8-bit attribute	[30][04]	[30][04]	attribute 0x04
0x31	16-bit attribute	[31][00][34][12]	[31][34][12]	attribute 0x1234
0x34	electronic key	[34][04] [12][00] [EF][BE] [0E][0B] [83] [01]	[34][04] [12][00] [EF][BE] [0E][0B] [83] [01]	vendor ID = 0x0012 product type = 0xBEEF product code = 0x0B0E major revision = 0x03 minor revision = 0x01 accept compatible keys = yes

4.1.9.5 Network segment

4.1.9.5.1 Network segment structure

The segment type (first octet) of a network segment shall be in the range 0x40 through 0x5F as shown in Table 116 (the most significant bits shall be 010). The network segment shall be used to specify network parameters which may be required by a node to transmit a message across a network. The network segment shall immediately precede the port segment of the device to which it applies. In other words, the network segment shall be the first item in the path that the device receives.

Table 116 – Network segments

First octet	Network segment name
0x40	reserved
0x41	schedule segment
0x42	fixed tag segment
0x43	production inhibit time segment
0x44 – 0x4F	reserved
0x050	safety segment
0x051 – 0x05E	reserved
0x5F	extended network segment

4.1.9.5.2 Schedule Segment

The segment type of the schedule network segment shall be 0x41. The schedule network segment shall specify

- a multiplier that, when multiplied by the NUT, gives the CM_API (actual packet interval) of the scheduled transport;
- a phase that determines on which values of the `Moderator.interval_count` to transmit.

NOTE 1 The data-link layer defines the Moderator.interval_count, see IEC 61158-4-2.

This segment shall be included in the path to a device so that each intermediary node can schedule link transmit time for subsequent scheduled traffic. The multiplier shall be one of 1, 2, 4, 8, 16, 32, 64 or 128. The phase shall be in the range 0 through (multiplier – 1). The second octet of the schedule network segment shall encode both the multiple and phase by adding them together.

NOTE 2 If a transport produces every 64th NUT starting on `interval_count` = 17, then the encoded value is 17 + 64 = 81.

An encoded value of zero shall specify that the transport has not yet been given permission to use the scheduled priority. If a node receives a request for a scheduled connection in which either no schedule network segment exists, or the schedule network segment exists but the encoded value is 0, the node shall return general status = 0x01, and extended status = 0x0317.

NOTE 3 A connection originator is given permission to use the scheduled priority through the Scheduling object (IEC 61158-5-2).

4.1.9.5.3 Fixed Tag Segment

The segment type of the fixed tag network segment shall be 0x42. The fixed tag network segment shall specify the fixed tag which is to be used when sending an unconnected message. This segment subtype shall precede the port segment within the path. If the fixed tag segment is not present, then the default fixed tag shall be used.

NOTE 4 The fixed tag segment can be used within the path of the Unconnected_Send service (see 4.1.5.6) of the Connection Manager to send requests to the Keeper object via the Management UCMM (see IEC 61158-5-2).

4.1.9.5.4 Production Inhibit Time Segment

The segment type of the production inhibit time segment shall be 0x43. The second octet shall specify the production inhibit time, which is the minimum time, in milliseconds, between successive transmissions of connected data for the specified connection.

NOTE 5 For example, if a production inhibit time of 10 milliseconds is specified, new data shall be sent no sooner than 10 milliseconds after the previous data.

The production inhibit time segment shall be used only for change-of-state transport class 1 connections over TCP/IP links, to place a limit on the packet rate. The production inhibit time segment may be sent to an end or intermediate device.

When the production inhibit time segment is used, the RPI shall determine when to re-send data. The RPI shall be larger than the production inhibit time. If the RPI is smaller than the production inhibit time, the Forward_Open response shall be returned with error (status 0x01, extended status 0x11B). If the production inhibit time segment is omitted, the producer shall use a default production inhibit time of 1/4 the RPI. If the production inhibit time is 0, there shall be no limit on how fast data may be sent on the connection.

NOTE 6 The method for transmitting APDUs over Ethernet-TCP/IP is specified in 4.3 and 11.

4.1.9.5.5 Safety Segment

See IEC 61784-3-2 for format.

4.1.9.5.6 Extended Network Segment

The extended network segment allows for the definition of additional network segment subtypes. The first word of data is the extended network segment subtype. The structure of the extended network segment is shown in Figure 12.

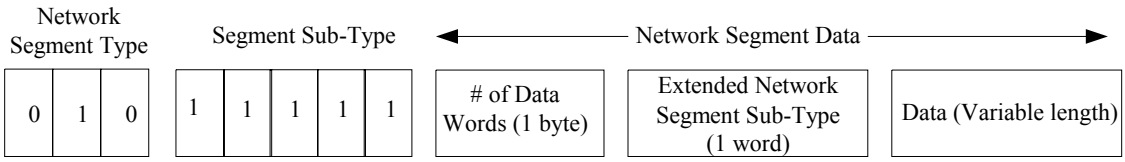


Figure 12 – Extended network segment

Each extended subtype defines the data that follows. The extended subtypes are enumerated in Table 117.

Table 117 – Extended subtype definitions

Extended Subtype Value	Extended Subtype Definition
0 – 65 535	Reserved for future use

4.1.9.6 Symbolic segment

The range of segment types reserved for symbolic segments shall be 0x60 through 0x7F.

4.1.9.7 Data segment

4.1.9.7.1 Data segment structure

The segment type (first octet) of a data segment shall be in the range 0x80 through 0x8F (the most significant bits shall be 1000). This standard only defines the format of the simple data segment with a segment type of 0x80; all other data segment types shall be reserved (0x81 through 0x8F).

4.1.9.7.2 Simple Data Segment

The segment type of the simple data segment shall be 0x80. The second octet shall represent the number of 16-bit words of variable length data. Following the second octet shall be the variable length data. A path may contain more than one simple data segment as shown in Table 118.

Table 118 – Data segment

Parameter	Format	Value
segment_type	USINT	always 0x80
segment_size	USINT	size of the data[] array in 16-bit words
data[]	UINT	

The data can be configuration information for an object, additional parameters necessary for the connection, or any other information. The data segment shall not be interpreted by any other device in the path, so only the originating and target applications need agree on its contents.

4.1.9.8 Extended symbol segment

4.1.9.8.1 Extended symbol segment structure

The segment type (first octet) of an extended symbol segment shall be in the range 0x90 through 0x9F (the most significant bits shall be 1001). This STANDARD only defines the format of the extended symbol segments 0x91. All other extended symbol segment types shall be reserved (0x90 and 0x92 through 0x9F).

4.1.9.8.2 ANSI Extended Symbol Segment

The segment type of the ANSI extended symbol segment shall be 0x91. The second octet shall represent the number of characters (8-bit) in the symbol. Following the second octet shall be the variable length symbol as shown in Table 119.

Table 119 – ANSI_Extended_Symbol segment

Parameter	Format	Value
segment_type	USINT	always 0x91
symbol_size	USINT	size of the symbol[] array
symbol[]	USINT	
pad	USINT	only present if symbol_size is odd, always set to zero

The `symbol_size` shall be the size of the `symbol[]` array in octets and shall not be zero. The `symbol_size` shall not count the `pad` octet if it is included.

4.1.10 Class, attribute and service codes

4.1.10.1 Code ranges

4.1.10.1.1 Defined ranges

There shall be three categories for address ranges of Class IDs, Attribute IDs and Service Codes as specified in Table 120.

Table 120 – Addressing categories

This category	Refers to
Open	A value which has the same meaning for all implementers. All object classes and services defined in IEC 61158-5-2 fall under this category.
Vendor-specific	A range of values specific to the vendor of a device. These are used by vendors to extend their devices beyond the available *Open* options. A vendor internally manages the use of values within this range. Applies to object classes, attributes, and services.
Object-class specific	A range of values whose meaning is defined by an object class. This range applies to Service Code definitions.

The Class ID, Attribute ID and Service code values shall be as defined in Table 121 through Table 123.

4.1.10.1.2 Class code ID ranges

The Class ID values shall be as shown in Table 121. Class Code = 0x00 is reserved and shall not be used.

Table 121 – Class code ID ranges

Range (hex)	Range (decimal)	Meaning	Quantity
00	00	Reserved	1
01 - 63	01 - 99	Open	99
64 - C7	100 - 199	Vendor Specific	100
C8 - EF	200 - 239	Reserved	40
F0 – 2FF	240 – 767	Open	528
300 - 4FF	768 - 1 279	Vendor Specific	512
500 - FFFF	1 280 - 65 535	Reserved	64 256

4.1.10.1.3 Attribute ID ranges

The Attribute ID values shall be as shown in Table 122. Attribute ID = 0x00 is reserved and shall not be used.

Table 122 – Attribute ID ranges

Range (hex)	Range (decimal)	Meaning	Quantity
00 – 63	00 – 99	Open	100
64 – C7	100 – 199	Vendor Specific	100
C8 – FF	200 – 255	Reserved	56
100 – 2FF	240 – 767	Open	512
300 - 4FF	768 – 1 279	Vendor Specific	512
500 – 8FF	1 280 – 2 303	Open	1 024
900 – CFF	2 304 – 3 327	Vendor Specific	1 024
D00 - FFFF	3 328 – 65 535	Reserved for future use	62 208

4.1.10.1.4 Service code ranges

The **Service code** shall be a unique hexadecimal value assigned to each service. The Service Code values shall be as shown in Table 123. Service Code = 0x00 is reserved and shall not be used. The object-specific service codes shall be unique only within the class which define them.

Table 123 – Service code ranges

Range (hex)	Range (decimal)	Meaning	Quantity
00	00	Reserved	1
01 - 31	01 - 49	Open. These are referred to as *Common Services*. They are defined in Type 2 clauses in IEC 61158-5-2	49
32 - 4A	50 - 74	Vendor Specific	25
4B - 63	75 - 99	Object Class Specific	25
64 - 7F	100 - 127	Reserved for future use	28
80 - FF	128 - 255	Reserved for response messages	128

4.1.10.2 Code definitions

4.1.10.2.1 Communication object classes

Table 124 defines the class codes for the object classes. All other class codes listed in within the “open” range and not listed in Table 124 shall be reserved. The requirements for these objects are defined in IEC 61158-5-2.

Table 124 – Class codes

Class code (hex)	Class code (decimal)	Object name
0x01	1	Identity object
0x02	2	Message Router object
0x03	3	DeviceNet object[a]
0x04	4	Assembly object
0x05	5	Connection object
0x06	6	Connection Manager object
0x2B	43	Acknowledge Handler object
0x43	67	Time Sync object
0xF0	240	ControlNet object[a]
0xF1	241	Keeper object[a]
0xF2	242	Scheduling object[a]
0xF3	243	Connection Configuration object[a]
0xF5	245	TCP/IP Interface object[a]
0xF6	246	Ethernet Link object[a]

[a] This object class is part of system management.

4.1.10.2.2 Predefined class attributes

Seven predefined Class Attribute IDs shall be reserved, and these predefined/reserved class attributes shall have the definitions listed in Table 125. Because these attributes are reserved, class Attribute ID numbers 1 through 7 shall always be reserved. Therefore, if a class attribute is added to an object class specification, it shall start with Attribute ID #8.

Table 125 – Reserved class attributes for all object class definitions

Attribute ID	Name	Data type	Semantics of values
1	Revision	UINT	The starting value assigned to this attribute is one (1). If updates that require an increase in this value are made, then the value of this attribute increases by one (1).
2	Max Instance	UINT or UDINT	The class definition shall define the data type used.
3	Number of Instances	UINT or UDINT	The class definition shall define the data type used.
4	Optional attribute list	STRUCT of	
	Number of attributes	UINT	
	Optional attributes	ARRAY of UINT	
5	Optional service list	STRUCT of	

Attribute ID	Name	Data type	Semantics of values
	Number services	UINT	
	Optional services	ARRAY of UINT	
6	Maximum ID Number Class Attributes	UINT	
7	Maximum ID Number Instance Attributes	UINT	

4.1.10.2.3 OM_Service codes

Codes of the common services for the deterministic control network objects shall be as shown in Table 126.

NOTE Table 126 lists the codes and names of the common services while IEC 61158-5-2 provides a general description of each service.

Table 126 – Common services list

Common service code	Common service name
0x00	reserved
0x01	Get_Attribute_All
0x02	Set_Attribute_All
0x03	Get_Attribute_List
0x04	Set_Attribute_List
0x05	Reset
0x06	Start
0x07	Stop
0x08	Create
0x09	Delete
0x0A – 0x0C	reserved
0x0D	Apply_Attributes
0x0E	Get_Attribute_Single
0x0F	reserved
0x10	Set_Attribute_Single
0x11	Find_Next_Object_Instance
0x12 – 0x14	reserved
0x15	Restore
0x16	Save
0x17	NOP (No operation)
0x18 – 0x1B	reserved
0x1C	Group_Sync
0x1D – 0x31	reserved

Codes of the object specific services for the Acknowledge Handler object shall be as shown in Table 127.

Table 127 – Acknowledge Handler object specific services list

Object specific service code	Common service name
0x4B	Add_AckData_Path
0x4C	Remove_AckData_Path

Codes of the object specific services for the Time Sync object shall be as shown in Table 128.

Table 128 – Time Sync object specific services list

Object specific service code	Common service name
0x4A	Management_Message
0x4B	Initialize

4.1.10.2.4 CM_Service codes

Codes of the services specific to the Connection Manager shall be as shown in Table 129.

NOTE Table 129 lists the codes and names of the services while IEC 61158-5-2 provides a general description of each service.

Table 129 – Services specific to Connection Manager

Service code	Service name
0x4E	Forward_Close
0x52	Unconnected_Send
0x54	Forward_Open
0x56	Get_Connection_Data
0x57	Search_Connection_Data
0x5A	Get_Object_Owner
0x5B	Large_Forward_Open

4.1.10.2.5 CO_Service codes

Codes of the services specific to the Connection object shall be as shown in Table 130.

NOTE Table 130 lists the codes and names of the services while IEC 61158-5-2 provides a general description of each service.

Table 130 – Services specific to Connection object

Service code	Service name
0x4B	Connection_Bind
0x4C	Producing_Application_Lookup
0x4E	SafetyClose
0x54	SafetyOpen

4.1.10.3 Device types

In order to allow interoperability and interchangeability, a device type shall be used to identify similar devices which

- exhibit the same behaviour;

- produce and/or consume the same set of data;
- contain the same set of configurable attributes.

The formal definition of this information is known as a Device Profile : all devices with the same device type number shall meet the requirements specified in the Device Profile for that device type.

“Device Type” is a required instance attribute of the Identity object as described in IEC 61158-5-2.

Device types shall be either publicly defined or vendor specific. Table 131 reveals the numbering scheme to be used for device type numbering.

Table 131 – Device type numbering

Range (hex)	Type	Quantity
00 - 63	Publicly Defined – Reserved	100
64 - C7	Vendor Specific	100
C8 - FF	Reserved	56
100 - 2FF	Publicly Defined – Reserved	512
300 - 4FF	Vendor Specific	512
500 - FFFF	Reserved	64 256

All publicly defined device types shall have the same meaning for all implementers and are reserved. The Generic Device type (type number = 0x00) shall define a device that does not fit into any of the defined device types.

NOTE The actual definition of publicly defined device types and corresponding Device Profiles is outside the scope of this standard.

Vendor specific device types may be developed and vendors need not publish them. Each vendor shall maintain their own vendor-specific device types and corresponding number allocation.

4.1.11 Error codes

4.1.11.1 CM errors

The error codes are returned with the response to a Connection Manager Service Request which resulted in an error. These error codes shall be used to help diagnose the problem with a Service Request. The error code shall be split into an 8 bit general status and one or more 16 bit words of extended status. Unless specified otherwise, only the first word of extended status shall be required. Additional words of extended status may be used to specify additional module specific debug information. All devices which originate messages shall be able to handle multiple words of extended status.

Table 132 provides a summary of the available error codes.

Table 132 – Connection Manager service request error codes

General Status	Extended Status	Explanation
0x00		Service completed successfully.
0x01	0x0000 – 0x00FF	Obsolete
0x01	0x0100	CONNECTION IN USE OR DUPLICATE FORWARD OPEN This extended status code shall be returned when an originator is trying to make a connection to a target with which the originator may have already established a connection (duplicate Forward_Open – see 4.1.5.3).
0x01	0x0101 – 0x0102	Reserved
0x01	0x0103	TRANSPORT CLASS AND TRIGGER COMBINATION NOT SUPPORTED A transport class and trigger combination has been specified which is not supported by the target. Routers shall not fail the connection based on the transport class and trigger combination. Only targets shall return this extended status code
0x01	0x0104 – 0x0105	Reserved
0x01	0x0106	OWNERSHIP CONFLICT The connection cannot be established since another connection has exclusively allocated some of the resources required for this connection. An example of this would be that only one exclusive owner connection can control an output point on an I/O Module. If a second exclusive owner connection (or redundant owner connection) is attempted, this error shall be returned. This extended status code shall only be returned by a target node.
0x01	0x0107	TARGET CONNECTION NOT FOUND This extended status code shall be returned in response to the forward_close request, when the connection that is to be closed is not found at the target node. This extended status code shall only be returned by a target node. Routers shall not generate this extended status code. If the specified connection is not found at the intermediate node, the close request shall still be forwarded using the path specified in the Forward_Close request.
0x01	0x0108	INVALID NETWORK CONNECTION PARAMETER This extended status code shall be returned as the result of specifying a connection type, connection priority, redundant owner or fixed/variable that is not supported by the target application. Only a target node shall return this extended status code.
0x01	0x0109	INVALID CONNECTION SIZE This extended status code is returned when the target or router does not support the specified connection size. This could occur at a target because the size does not match the required size for a fixed size connection. It could occur at a router if the requested size is too large for the specified network. An additional status may follow indicating the maximum connection size supported by the responding node. The additional status word is required when issued in response to the Large_Forward_Open.
0x01	0x010A – 0x010F	Reserved
0x01	0x0110	TARGET FOR CONNECTION NOT CONFIGURED This extended status code shall be returned when a connection is requested to a target application that has not been configured and the connection request does not contain a data segment for configuration (see 4.1.9.7). Only a target node shall return this extended status code.

General Status	Extended Status	Explanation
0x01	0x0111	RPI NOT SUPPORTED. This extended status code shall be returned if the device can not support the requested O⇒T or T⇒O RPI. This extended status code shall also be used if the connection time-out multiplier produces a time-out value that is not supported by the device or the production inhibit time is not valid.
0x01	0x0112	Reserved
0x01	0x0113	OUT OF CONNECTIONS Connection Manager cannot support any more connections. The maximum number of connections supported by the Connection Manager has already been created.
0x01	0x0114	VENDOR ID OR PRODUCT CODE MISMATCH The Product Code or Vendor Id specified in the electronic key logical segment does not match the Product Code or Vendor Id of the target device.
0x01	0x0115	PRODUCT TYPE MISMATCH The Product Type specified in the electronic key logical segment does not match the Product Type of the target device.
0x01	0x0116	REVISION MISMATCH The major and minor revision specified in the electronic key logical segment does not correspond to a valid revision of the target device.
0x01	0x0117	INVALID PRODUCED OR CONSUMED APPLICATION PATH The produced or consumed application path specified in the connection path does not correspond to a valid produced or consumed application path within the target application. This error could also be returned if a produced or consumed application path was required, but not provided by a connection request.
0x01	0x0118	INVALID OR INCONSISTENT CONFIGURATION APPLICATION PATH An application path specified for the configuration data does not correspond to a configuration application or is inconsistent with the consumed or produced application paths. For example the connection path specifies float configuration data while the produced or consumed paths specify integer data.
0x01	0x0119	NON-LISTEN ONLY CONNECTION NOT OPENED Connection request fails since there are no non-listen only connection types currently open. See IEC 61158-5-2, 6.3.1.4.5, for a description of application connection types. The extended status code shall be returned when an attempt is made to establish a listen only connection type to a target, which has no non-listen only connection already established.
0x01	0x011A	TARGET OBJECT OUT OF CONNECTIONS The maximum number of connections supported by this instance of the target object has been exceeded. For example, the Connection Manager could support 20 connections while the target object can only support 10 connections. On the 11th Connection Request to the target object, this extended status code would be used to signify that the maximum number of connections already exist to the target object.
0x01	0x011B	RPI IS SMALLER THAN THE PRODUCTION INHIBIT TIME The Target to Originator RPI is smaller than the Target to Originator Production Inhibit Time.
0x01	0x011C – 0x0202	Reserved

General Status	Extended Status	Explanation
0x01	0x0203	CONNECTION TIMED OUT This extended status code shall occur when a client tries to send a connected message over a connection that has been timed-out. This extended status code shall only occur locally at the producing node.
0x01	0x0204	UNCONNECTED REQUEST TIMED OUT The Unconnected Request Timed Out error shall occur when the UCMM times out before a response is received. This may occur for an Unconnected_Send, Forward_Open, or Forward_Close service. This typically means that the UCMM has tried a link specific number of times using a link specific retry timer and has not received an acknowledgement or response. This may be the result of congestion at the destination node or may be the result of a node not being powered up or present. This extended status code shall be returned by the originating node or any intermediate node.
0x01	0x0205	PARAMETER ERROR IN UNCONNECTED REQUEST SERVICE For example, this shall be caused by a Connection Tick Time (see IEC 61158-5-2, 6.2.3.2.1.7) and Connection time-out combination in an Unconnected_Send, Forward_Open, or Forward_Close service that is not supported by an intermediate node.
0x01	0x0206	MESSAGE TOO LARGE FOR UNCONNECTED_SEND SERVICE This shall be caused when the Unconnected_Send is too large to be sent out on a network.
0x01	0x0207	UNCONNECTED ACKNOWLEDGE WITHOUT RESPONSE The message was sent via the unconnected message service and an acknowledge was received but a data response message was not received.
0x01	0x0208 – 0x0300	Reserved
0x01	0x0301	NO BUFFER MEMORY AVAILABLE The extended status code shall occur when insufficient connection buffer memory is available in the target or any router devices. Routers and target nodes shall return this error.
0x01	0x0302	LINK TRANSMIT TIME NOT AVAILABLE FOR DATA This extended status code shall be returned by any device in the path that is a producer and can not allocate sufficient link transmit time for the connection on its link. This can occur at any node. This can only occur for connections that are specified as scheduled priority.
0x01	0x0303	NO CONSUMED CONNECTION ID FILTER AVAILABLE Any device in the path that contains a link consumer for the connection and does not have an available consumed_connection_id filter available shall return this extended status code.
0x01	0x0304	NOT CONFIGURED TO SEND SCHEDULED PRIORITY DATA If requested to make a connection that specifies scheduled priority, any device that is unable to send packets during the scheduled portion of the network update time interval shall return this extended status code. For example, on CP 2/1 this code shall be returned by a node whose MAC ID is greater than maximum scheduled node (SMAX).
0x01	0x0305	SCHEDULE SIGNATURE MISMATCH This extended status code shall be returned when the connection scheduling information in the originator device is not consistent with the connection scheduling information on the target network.
0x01	0x0306	SCHEDULE SIGNATURE VALIDATION NOT POSSIBLE This extended status code shall be returned when the connection scheduling information in the originator device can not be validated on the target network. For example, on CP 2/1 this code shall be returned when there is no Keeper in the master state.

General Status	Extended Status	Explanation
0x01	0x0307 – 0x0310	Reserved
0x01	0x0311	PORT NOT AVAILABLE A Port specified in a Port Segment is Not Available or does not exist.
0x01	0x0312	LINK ADDRESS NOT VALID Link Address specified in Port Segment Not Valid This extended status code is the result of a port segment that specifies a link address that is not valid for the target network type. This extended status code shall not be used for link addresses that are valid for the target network type but do not respond.
0x01	0x0313 – 0x0314	Reserved
0x01	0x0315	INVALID SEGMENT IN CONNECTION PATH Invalid Segment Type or Segment Value in Connection Path This extended status code is the result of a device being unable to decode the connection path. This could be caused by an unrecognised path type, a segment type occurring unexpectedly, or a myriad of other problems in the connection path.
0x01	0x0316	ERROR IN FORWARD CLOSE SERVICE CONNECTION PATH The path in the Forward_Close service does not match the connection being closed. This means the connection points to a different module or application than is specified in the path. The connection is deleted but the error message shall be returned.
0x01	0x0317	SCHEDULING NOT SPECIFIED Either the Schedule Network Segment was not present or the Encoded Value in the Schedule Network Segment is invalid (0).
0x01	0x0318	LINK ADDRESS TO SELF INVALID Under some conditions (depends on the device), a link address in the Port Segment which points to the same device (loopback to yourself) is invalid.
0x01	0x0319	SECONDARY RESOURCES UNAVAILABLE In a dual chassis redundant system, a connection request that is made to the primary system shall be duplicated on the secondary system. If the secondary system is unable to duplicate the connection request, then this extended status code shall be returned.
0x01	0x031A	RACK CONNECTION ALREADY ESTABLISHED A request for a module connection has been refused because part of the corresponding data is already included in a rack connection.
0x01	0x031B	MODULE CONNECTION ALREADY ESTABLISHED A request for a rack connection has been refused because part of the corresponding data is already included in a module connection.
0x01	0x031C	MISCELLANEOUS This extended status is returned when no other extended status code applies for a connection related error.

General Status	Extended Status	Explanation
0x01	0x031D	REDUNDANT CONNECTION MISMATCH This extended status code shall be returned when the following fields do not match when attempting to establish a redundant owner connection to the same target path: O->T_RPI; O->T_connection_parameters; T->O_RPI; T->O_connection_parameters; xport_type_and_trigger.
0x01	0x031E	NO MORE USER CONFIGURABLE LINK CONSUMER RESOURCES AVAILABLE IN THE PRODUCING MODULE A target shall return this extended status when the configured number of consumers for a producing application are already in use.
0x01	0x031F	NO MORE USER CONFIGURABLE LINK CONSUMER RESOURCES AVAILABLE IN THE PRODUCING MODULE A target shall return this extended status when there are no consumers configured for a producing application to use.
0x01	0x0320 – 0x07FF	Vendor specific
0x01	0x0800	Network link in path to module is offline
0x01	0x0801 – 0x080F	Reserved
0x01	0x0810	NO TARGET APPLICATION DATA AVAILABLE This extended status code is returned when the target application does not have valid data to produce for the requested connection. Only the target side of a connection shall return this extended status code.
0x01	0x0811	NO ORIGINATOR APPLICATION DATA AVAILABLE This extended status code is returned when the originator application does not have valid data to produce for the requested connection. Only the originator side of a connection shall indicate this extended status code.
0x01	0x0812	Reserved
0x01	0x0813	NOT CONFIGURED FOR OFF-SUBNET MULTICAST A multicast connection has been requested between a producer and a consumer that are on different subnets, and the producer is not configured for off-subnet multicast.
0x01	0x0814 – 0xFCFF	Reserved
0x09	Index to Element	ERROR IN DATA SEGMENT. This general status code shall be returned when there is an error in the data segment of a forward open. The Extended Status shall be the index to where the error was encountered in the Data Segment (see 4.1.9.7).
0x0C	Optional	OBJECT STATE ERROR This general status code shall be returned when the state of the target object of the connection prevents the service request from being handled. The Extended Status reports the object's present state. The extended status is optional. For example, a target (application) object of the connection may need to be in an edit mode before attributes can be set. This is different from a service being rejected due to the state of the device.

General Status	Extended Status	Explanation
0x10	Optional	DEVICE STATE ERROR This general status code shall be returned when the state of the device prevents the service request from being handled. The Extended Status reports the device's present state. The extended status is optional. For example, a controller may have a key switch which when set to the "hard run" state causes Service Requests to several different objects to fail (i.e. program edits). This general status code would then be returned.

NOTE 1 The word “n/a” in the Extended Status Column is used to signify that there is no additional Extended Status which is required to be returned for the particular General Status Code.

NOTE 2 The word “optional” in the Extended Status Column is used to signify that if Extended Status information is used, then the first word of that extended status is already defined and user defined extended status shall begin with the second word of extended status.

4.1.11.2 OM errors

4.1.11.2.1 General status format

General status used in the service primitives is specified in Table 133.

Table 133 – General status codes

Status code	Name	Description and meaning of status code
0x00	Success	Service was successfully performed by the object specified
0x01	Connection failure	A connection related service failed along the connection path
0x02	Resource unavailable	Resources needed for the object to perform the requested service were unavailable
0x03	Invalid parameter value	See status code 0x20, which is the preferred value to use for this condition
0x04	Path segment error	The path segment identifier or the segment syntax was not understood by the processing node. Path processing shall stop when a path segment error is encountered
0x05	Path destination unknown	The path is referencing an object class, instance or structure element that is not known or is not contained in the processing node. Path processing shall stop when a path destination unknown error is encountered
0x06	Partial transfer	Only part of the expected data was transferred
0x07	Connection lost	The messaging connection was lost
0x08	Service not supported	The requested service was not implemented or was not defined for this class or object instance
0x09	Invalid attribute value	Invalid attribute data detected
0x0A	Attribute list error	An attribute in the Get_Attribute_List or Set_Attribute_List response has a non zero status
0x0B	Already in requested mode/state	The object is already in the mode/state being requested by the service
0x0C	Object state conflict	The object cannot perform the requested service in its current mode/state
0x0D	Object already exists	The requested instance of object to be created already exists
0x0E	Attribute not settable	A request to modify a non-modifiable attribute was received
0x0F	Privilege violation	A permission/privilege check failed
0x10	Device state conflict	The device’s current mode/state prohibits the execution of the requested service

Status code	Name	Description and meaning of status code
0x11	Response data too large	The data to be transmitted in the response buffer is larger than the allocated response buffer
0x12	Fragmentation of a primitive value	The service specified an operation that is going to fragment a primitive data value, i.e. half a REAL data type
0x13	Not enough data	The service did not supply enough data to perform the specified operation
0x14	Attribute not supported	The attribute specified in the request is not supported
0x15	Too much data	The service supplied more data than was expected
0x16	Object does not exist	The object specified does not exist in the device
0x17	Service fragmentation sequence not in progress	The fragmentation sequence for this service is not currently active for this data
0x18	No stored attribute data	The attribute data of this object was not saved prior to the requested service
0x19	Store operation failure	The attribute data of this object was not saved due to a failure during the attempt.
0x1A	Routing failure, request packet too large for network	The service request packet was too large for transmission on a network in the path to the destination. The routing device was forced to abort the service
0x1B	Routing failure, response packet too large for network	The service response packet was too large for transmission on a network in the path from the destination. The routing device was forced to abort the service
0x1C	Missing attribute list entry data	The service did not supply an attribute in a list of attributes that was needed by the service to perform the requested behaviour
0x1D	Invalid attribute value list	The service is returning the list of attributes supplied with status information for those attributes that were invalid
0x1E	Embedded service error	An embedded service resulted in an error
0x1F	Vendor specific error	A vendor specific error has been encountered. The extended status code field of the error response defines the particular error encountered. Use of this general status code should only be performed when none of the status codes presented in this table or within an object class definition accurately reflect the error
0x20	Invalid parameter	A parameter associated with the request was invalid. This code is used when a parameter does not meet the requirements of this specification and/or the requirements defined in an application object specification
0x21	Write once value or medium already written	An attempt was made to write to a write once medium (e.g. WORM drive, PROM) that has already been written, or to modify a value that cannot be changed once established
0x22	Invalid Response Received	An invalid response is received (e.g. response service code does not match the request service code, or response message is shorter than the minimum expected reply size). This status code can serve for other causes of invalid responses
0x23 – 0x24	Reserved	Reserved for future extensions
0x25	Key Failure in path	The Key Segment which was included as the first segment in the path does not match the destination module. The object specific status shall indicate which part of the key check failed.
0x26	Path Size Invalid	The size of the path which was sent with the Service Request is either not large enough to allow the Request to be routed to an object or too much routing data was included.
0x27	Unexpected attribute in list	An attempt was made to set an attribute that cannot be set at this time.
0x28	Invalid Member ID	The Member ID specified in the request does not exist in the specified Class/Instance/Attribute.
0x29	Member not settable	A request to modify a non-modifiable member was received.

Status code	Name	Description and meaning of status code
0x2A	Group 2 only server general failure	This status code may only be reported by CP 2/3 Group 2 Only servers with 4 koctets or less code space and only in place of Service not supported, Attribute not supported and Attribute not settable
0x2B – 0xCF	Reserved	Reserved for future extensions
0xD0 – 0xFF	Reserved for object class and service errors	This range of status codes shall be used to indicate object class specific errors. Use of this range should only be performed when none of the status codes presented in this table accurately reflect the error that was encountered

NOTE IEC 61158-5-2 contains more detail on the service response general status codes for each common service.

4.1.11.2.2 Extended status format

The MR_response_Header contains a parameter called Extended_status[] which is labelled extended status data.

Actual usage and definition of this extended status is dependant on the object class or service: each object class defines its own extended status values and value ranges (including the vendor specific ones).

4.1.11.2.3 Object-specific general and extended status format

4.1.11.2.3.1 General

The following clauses specify the format of object-specific general and extended status.

4.1.11.2.3.2 Identity object status format

Table 134 specifies the format of object-specific general and extended status for the Identity object.

Table 134 – Identity object status codes

General status code	8-bit associated extended status codes	Status Name	Description of Status
0x00 – 0xCF		General status codes	Defined in 4.1.11.2.1
	0x00 – 0xEE		Reserved extended status codes
	0xF0 – 0xFE	Vendor specific	Vendor specific extended status codes
	0xFF		Used with all general status codes when required and no other extended status code is assigned
0xD0		Hardware diagnostic	Device self testing and hardware diagnostic conditions
	0x00		Reserved
	0x01		Checksum (or CRC) error – Code space/ROM – Boot section
	0x02		Checksum (or CRC) error – Code space/ROM – Application section
	0x03		Checksum (or CRC) error – NV (flash/EEPROM) memory
	0x04		Invalid non-volatile (NV) memory – Configuration bad
	0x05		Invalid non-volatile (NV) memory – No configuration established
	0x06		RAM memory bad – The RAM memory in the device was determined to be experiencing inoperative cells

General status code	8-bit associated extended status codes	Status Name	Description of Status
	0x07		ROM/Flash memory bad
	0x08		Flash/EEPROM (NV) Memory Bad
	0x09		Interconnect wiring error / signal path problem
	0x0A		Power problem – Over current
	0x0B		Power problem – Over voltage
	0x0C		Power problem – Under voltage
	0x0D		Internal sensor problem
	0x0E		System clock fault
	0x0F		Hardware configuration does not match NV configuration
	0x10		Watchdog disabled/idle
	0x11		Watchdog timer expired
	0x12		Device over temperature
	0x13		Ambient temperature outside of operating limits
	0x14 – 0xEF	(reserved)	Reserved
	0xF0 – 0xFE		Vendor specific extended status codes
	0xFF		Used with all general status codes when required and no other extended status code is assigned
0xD1		Device status/states	Device status events and conditions
	0x01		Power applied
	0x02		Device reset
	0x03		Device power loss
	0x04		Activated
	0x05		Deactivated
	0x06		Enter self-test state
	0x07		Enter standby state
	0x08		Enter operational state
	0x09		Non-specific minor recoverable fault detected
	0x0A		Non-specific minor unrecoverable fault detected
	0x0B		Non-specific major recoverable fault detected
	0x0C		Non-specific major unrecoverable fault detected
	0x0D		Fault(s) corrected
	0x0E		Ccv changed
	0x0F		Heartbeat interval changed
	0x10 – 0xEF	(reserved)	
	0xF0 – 0xFE	Vendor specific	Vendor specific
	0xFF		Used with all general status codes when required and no other extended status code is assigned
0xD2 – 0xEF		Object specific general status codes	Reserved – Not yet assigned
	0x00 – 0xFF	Reserved	

General status code	8-bit associated extended status codes	Status Name	Description of Status
0xF0 – 0xFF		Vendor specific general status codes	A vendor specific error has been encountered. The extended status code field of the error response defines the particular error encountered. Use of this general status code should only be performed when none of the status codes presented in this table or within an object class definition accurately reflect the error
	0x00 – 0xFF	Vendor specific extended status codes	All extended status codes are available for association with each vendor specific general status code

4.2 Data abstract syntax specification

4.2.1 Transport format specification

The lower layers of open system architectures are concerned with the transport of user data among distributed functional units. In these layers, the user data may be regarded simply as a sequence of octets. However, application layer entities may manipulate the values of quite complex data types. To achieve independence between the application layer and lower layers, data types may be specified in an abstract syntax notation.

Supplementing the abstract syntax with one or more algorithms (called encoding rules) may determine the values of the lower layer octets which carry the application layer values. The combination of the abstract syntax with a single set of transfer rules produces a specific transfer syntax.

4.2.2 Abstract syntax notation

The data type definitions provided in this STANDARD shall be written in Abstract Syntax Notation One (ASN.1), as defined in ISO/IEC 8824. These type definitions shall be a part of the ASN.1 module "Network DataTypes." The beginning ASN.1 statement indicating that these definitions are in this module is:

```
control network DataTypes DEFINITIONS ::= BEGIN
```

and the closing ASN.1 statement shall be the keyword "END".

The abstract definitions that follow shall comprise the set of control network data types. In addition, provision is made to extend or derive new data types based on existing defined types, and to include those in a "type dictionary."

4.2.3 Control network data specification

The notation [typeId] for directly derived, enumerated, subrange and structured bit string data shall mean that the tag shall be taken from the "type" field in the corresponding VariableDictionaryEntry.

```
Network Data ::= CHOICE{ElementaryData, DerivedData}

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

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

DirectlyDerivedData ::= [typeId] NetworkData

EnumeratedData ::= [typeId] USINT

SubrangeData ::= [typeId] FixedLengthInteger

StructuredBitStringData ::= [typeId] FixedLengthBitString

FixedLengthInteger ::= CHOICE {SignedInteger, UnsignedInteger}

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

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

FixedLengthReal ::= CHOICE {REAL, LREAL}

AnyTime ::= CHOICE {ITIME,TIME, FTIME, LTIME}

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

AnyString ::= CHOICE {STRING, STRING2}

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

BOOL ::= [PRIVATE 1] IMPLICIT BOOLEAN

SINT ::= [PRIVATE 2] IMPLICIT OCTET STRING-- 1 octet

INT ::= [PRIVATE 3] IMPLICIT OCTET STRING-- 2 octets

DINT ::= [PRIVATE 4] IMPLICIT OCTET STRING-- 4 octets

LINT ::= [PRIVATE 5] IMPLICIT OCTET STRING-- 8 octets

USINT ::= [PRIVATE 6] IMPLICIT OCTET STRING-- 1 octet

UINT ::= [PRIVATE 7] IMPLICIT OCTET STRING-- 2 octets

UDINT ::= [PRIVATE 8] IMPLICIT OCTET STRING-- 4 octets

ULINT ::= [PRIVATE 9] IMPLICIT OCTET STRING-- 8 octets

REAL ::= [PRIVATE 10] IMPLICIT OCTET STRING-- 4 octets

LREAL ::= [PRIVATE 11] IMPLICIT OCTET STRING-- 8 octets

STIME ::= [PRIVATE 12] IMPLICIT DINT

DATE ::= [PRIVATE 13] IMPLICIT UINT

TIME_OF_DAY ::= [PRIVATE 14] IMPLICIT UDINT

DATE_AND_TIME ::= [PRIVATE 15] IMPLICIT SEQUENCE {
 time_of_day  UDINT,
 date  UINT }

STRING ::= [PRIVATE 16] IMPLICIT SEQUENCE {
            charcount      UINT,
            stringcontents OCTET STRING} -- one octet per character

SWORD ::= [PRIVATE 17] IMPLICIT OCTET STRING-- 1 octet

WORD ::= [PRIVATE 18] IMPLICIT OCTET STRING-- 2 octets

DWORD ::= [PRIVATE 19] IMPLICIT OCTET STRING-- 4 octets

LWORD ::= [PRIVATE 20] IMPLICIT OCTET STRING-- 8 octets

STRING2 ::= [PRIVATE 21] IMPLICIT SEQUENCE {
            charcount      UINT,
            string2contents OCTET STRING} -- 2 octets/ character
```

```
FTIME ::= [PRIVATE 22] IMPLICIT DINT

LTIME ::= [PRIVATE 23] IMPLICIT LINT

ITIME ::= [PRIVATE 24] IMPLICIT INT

STRINGN ::= [PRIVATE 25] IMPLICIT SEQUENCE {
            charsize    UINT,
            charcount   UINT,
            stringNcontents OCTET STRING} -- N octets/ character

SHORT_STRING ::= [PRIVATE 26] IMPLICIT SEQUENCE {
            charcount      USINT,
            stringcontents OCTET STRING} -- one octet per character

TIME        ::=   [PRIVATE 27] IMPLICIT DINT

EPATH       ::=   [PRIVATE 28] IMPLICIT OCTET STRING  -- IEC 61158-6-2

STRINGI     ::=   [PRIVATE 30] IMPLICIT SEQUENCE{
            Stringnum   USINT (number of strings)
            array of    STRUCT
            language1   USINT (first character from ISO 639-2/T)
            language2   USINT (second character from ISO 639-2/T)
            language3   USINT (third character from ISO 639-2/T)
            datatype    EPATH limited to the values 0xD0,0xD5,0xD9,and
0xDA)
            charset     UINT     from IANA MIB Printer Codes (RFC 1759))
            stringcontents    CHOICE OF (SHORT_STRING, STRING, STRING2,
                        or STRINGN) -- based on datatype field

ARRAY ::= SEQUENCE OF NetworkData -- All of same base type

STRUCT ::= SEQUENCE OF NetworkData -- May be different types

FunctionBlockData ::= SET{
            inputs   [0] IMPLICIT STRUCT OPTIONAL,
            outputs      [1] IMPLICIT STRUCT OPTIONAL,
            controlInputs  [2] IMPLICIT STRUCT OPTIONAL,
            controlOutputs [3] IMPLICIT STRUCT OPTIONAL}
```

4.2.4 Data type specification / dictionaries

The definition of an object may include text that defines attributes. Attributes shall be assigned a *Data Type* in an object specification. The Data Type may be one of those defined in this standard or may be an object specific extension to this standard. The following definition shall provide a *Type Specification* for data and shall provide a structure for extending or deriving new data types based on existing defined types.

```
Dictionary ::= CHOICE {VariableDictionary, TypeDictionary}

VariableDictionary ::= SEQUENCE OF VariableDictionaryEntry

VariableDictionaryEntry ::= SEQUENCE{
            name  AnyString,
            id FixedLengthInteger,
            type  TypeID,
            ranges   SEQUENCE OF Subrange,-- for arrays
            accessPrivilege   BOOL {READ_ONLY(0), READ_WRITE(1)}
TypeID ::= OCTET STRING -- ASN.1 encoded tag value of the

-- DataTypeSpecification module

Subrange ::= SEQUENCE {
            minValue FixedLengthInteger,
            maxValue FixedLengthInteger}

TypeDictionary ::= SEQUENCE OF TypeDictionaryEntry
```

```
TypeDictionaryEntry ::= SEQUENCE {
            name  AnyString,
            type  TypeID,
            spec  DataTypeSpecification}
DataTypeSpecification ::= CHOICE {
            alt       [PRIVATE 0] IMPLICIT AlternateTypeSpec,
            bool   [PRIVATE 1] IMPLICIT NULL, -- BOOL
            sint   [PRIVATE 2] IMPLICIT NULL, -- SINT
            int    [PRIVATE 3] IMPLICIT NULL, -- INT
            dint   [PRIVATE 4] IMPLICIT NULL, -- DINT
            lint   [PRIVATE 5] IMPLICIT NULL, --    LINT
            usint  [PRIVATE 6] IMPLICIT NULL, --    USINT
            uint   [PRIVATE 7] IMPLICIT NULL, --    UINT
            udint  [PRIVATE 8] IMPLICIT NULL, --    UDINT
            ulint  [PRIVATE 9] IMPLICIT NULL, --    ULINT
            real   [PRIVATE 10]   IMPLICIT NULL, --    REAL
            lreal  [PRIVATE 11]   IMPLICIT NULL, --    LREAL
            stime  [PRIVATE 12]   IMPLICIT NULL, --    STIME
            date   [PRIVATE 13]   IMPLICIT NULL, --    DATE
            tod    [PRIVATE 14]   IMPLICIT NULL, --    TIME_OF_DAY
            dat    [PRIVATE 15]   IMPLICIT NULL, --    DATE_AND_TIME
            str1   [PRIVATE 16]   IMPLICIT NULL, --    STRING
            sword  [PRIVATE 17]   IMPLICIT NULL, --    SWORD
            word   [PRIVATE 18]   IMPLICIT NULL, --    WORD
            dword  [PRIVATE 19]   IMPLICIT NULL, --    DWORD
            lword  [PRIVATE 20]   IMPLICIT NULL, --    LWORD
            str2   [PRIVATE 21]   IMPLICIT NULL, --    STRING2
            ftime  [PRIVATE 22]   IMPLICIT NULL, --    FTIME
            ltime  [PRIVATE 23]   IMPLICIT NULL, --    LTIME
            itime  [PRIVATE 24]   IMPLICIT NULL, --    ITIME
            strN   [PRIVATE 25]   IMPLICIT NULL, --    STRINGN
            shstr  [PRIVATE 26]   IMPLICIT NULL, --    SHORT_STRING
            time      [PRIVATE 27]   IMPLICIT NULL, --    TIME
            epath     [PRIVATE 28]   IMPLICIT NULL, --    EPATH
            strI      [PRIVATE 30]   IMPLICIT NULL, -- STRINGI
            constructedData  CHOICE {
               abbrvStruc   [0]   IMPLICIT AbbreviatedStrucTypeSpec,
               abbrvArr     [1]   IMPLICIT AbbreviatedArrayTypeSpec,
               frmlStruc    [2]   IMPLICIT FormalStrucTypeSpec,
               frmlArr      [3]   IMPLICIT FormalArrayTypeSpec,
               expBitStr    [4]   IMPLICIT ExpandedFixedLenBitStrTypeSpec,
               expStr1      [5]   IMPLICIT ExpandedStringTypeSpec,
               expStr2      [6]   IMPLICIT ExpandedString2TypeSpec}
}

AbbreviatedStrucTypeSpec ::= UINT

AbbreviatedArrayTypeSpec ::= DataTypeSpecification

FormalStrucTypeSpec ::= SEQUENCE OF DataTypeSpecification
FormalArrayTypeSpec ::= SEQUENCE {
            lowBound [0] IMPLICIT FixedLengthInteger, -- Array Lower Bound
            highBound   [1] IMPLICIT FixedLengthInteger, -- Array Upper
Bound
            dataType    DataTypeSpecification }
```

```
ExpandedFixedLenBitStrTypeSpec ::= SEQUENCE {
            bitStrType DataTypeSpecification    -- SWORD, WORD, DWORD, or
LWORD
            bitFields [7] IMPLICIT BitFieldDef}

BitFieldDef ::= SEQUENCE OF {
            bitDef [2] IMPLICIT OCTET STRING}   -- Length is always 2
octets.
                                        -- First octet contains starting
                                        -- Bit Position. Trailing octet
                                        -- contains the number of bits.

ExpandedStringTypeSpec ::= UINT-- String Length In Octets

ExpandedString2TypeSpec ::= UINT-- String Length In Octets
AlternateTypeSpec ::= CHOICE {
            directlyDerivedTypeSpec [0]   IMPLICIT    TypeID,
            subrangeTypeSpec        [1]   IMPLICIT    SubrangeTypeSpec ,
            enumeratedTypeSpec      [2]   IMPLICIT    EnumeratedTypeSpec,
            fbTypeSpec              [3]   IMPLICIT    FBTypeSpec}

SubrangeTypeSpec ::= SEQUENCE{
            baseType    TypeID,     -- NOTE  minValue and maxValue
            minValue    FixedLengthInteger,  -- shall be within the range
            maxValue    FixedLengthInteger}  -- of baseType values

EnumeratedTypeSpec ::= SEQUENCE OF AnyString

BitNameDefintion ::= SEQUENCE {
            bitName  AnyString,
            bitNumber   USINT}
FBTypeSpec ::= SET{
            inputs          [0] IMPLICIT FbtElementSpec OPTIONAL,
            outputs         [1] IMPLICIT FbtElementTypeSpec OPTIONAL,
            controlInputs   [2] IMPLICIT FbtElementTypeSpec OPTIONAL,
            controlOutputs  [3] IMPLICIT FbtElementTypeSpec OPTIONAL}

FbtElementTypeSpec ::= SEQUENCE OF ElementSpec

ElementSpec ::= SEQUENCE {
            name     AnyString,
            typespec    ElementTypeSpec}

ElementTypeSpec ::= CHOICE {
            [0] IMPLICIT TypeID,
            [1] IMPLICIT SubrangeTypeSpec,
            [2] IMPLICIT EnumeratedTypeSpec,
            [3] IMPLICIT FormalArrayTypeSpec,
            [4] IMPLICIT ExpandedStringTypeSpec,
            [5] IMPLICIT ExpandedString2TypeSpec}
```

The following END statement shall terminate the ASN.1 module opened in 4.1.

END.

4.3 Encapsulation abstract syntax

4.3.1 Encapsulation protocol

4.3.1.1 Common note

NOTE In order to send TPDUs over TCP/IP, an encapsulation protocol is required. This subclause defines the encapsulation protocol requirements. The encapsulation protocol is a generic protocol that may be used for transporting data other than Type 2 TPDUs.

4.3.1.2 Encapsulation messages

All encapsulation messages shall be composed of a fixed-length header of 24 octets followed by an optional data portion. The total encapsulation message length (including header) shall be limited to 65 535 octets. The encapsulation message length shall not override length restrictions imposed by the encapsulated protocol. Its structure shall be as shown in Figure 13, where the top of the diagram indicates the portion of the message sent first on the wire.

NOTE The encapsulation data portion of the message is optional.

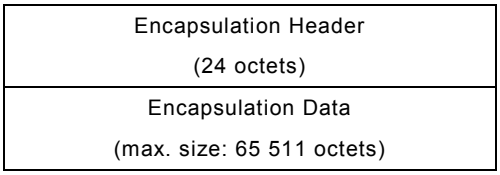


Figure 13 – Encapsulation message

4.3.1.3 Encapsulation header

The encapsulation header shall be as shown in Table 135.

Table 135 – Encapsulation header

Field name	Format	Description
Command	UINT	Encapsulation command
Length	UINT	Length, in octets, of the data portion of the message, i.e., the number of octets following the header
Session handle	UDINT	Session identification (application dependent)
Status	UDINT	Status code
Sender context	ARRAY of 8 USHORT	Information pertinent only to the sender of an encapsulation command
Options	UDINT	Options flags

Multi-octet integer fields in encapsulation messages shall be transmitted as specified in 5.

Although the header contains no explicit information to distinguish between a request and a reply, this information shall be determined in either of two ways:

- implicitly, by the command and the context in which the message is generated. (For example, in the case of the RegisterSession command, the request is generated by an originator and the target generates the reply) ;
- explicitly, by the contents of an encapsulated protocol packet in the data part of the message.

4.3.1.4 Command field

The allocation of command codes shall be as shown in Table 136.

Table 136 – Encapsulation command codes

Command code	Command
0x00	Nop
0x01 – 0x03	Reserved
0x04	ListServices
0x05 – 0x64	Reserved
0x65	RegisterSession
0x66	UnRegisterSession
0x67 – 0x6E	Reserved
0x6F	SendRRData
0x70	SendUnitData
0x71 – 0xFFFF	Reserved

A device shall accept commands that it does not support without breaking the session or underlying TCP connection. A status code indicating that an unsupported command was received shall be returned to the sender of the message (see 4.3.1.7).

4.3.1.5 Length field

The length field in the header shall specify the size in octets of the data portion of the message. The field shall contain zero for messages that contain no data. The total length of a message shall be the sum of the number contained in the length field plus the 24-octet size of the encapsulation header.

The entire encapsulation message shall be read even if the length may be invalid for a particular command or exceeds the target's internal buffers.

NOTE Failure to read the entire message can result in losing track of the message boundaries in the TCP octet stream.

4.3.1.6 Session handle field

The Session Handle shall be generated by the target and returned to the originator in response to a RegisterSession request. The originator shall insert it in all subsequent session requests (requests using session commands) to that particular target. In the case where the target initiates and sends a command to the originator, the target shall include this field in the request that it sends to the originator.

NOTE Some commands (i.e., Nop) do not require a session handle, even if a session has been established. Whether or not a particular command requires a session is noted in the description of that command.

4.3.1.7 Status field

The value in the Status field shall indicate whether or not the receiver was able to execute the requested encapsulation command. A value of zero in a reply shall indicate successful execution of the command. In all requests issued by the sender, the Status field shall contain zero. If the receiver receives a request with a non-zero Status field, the request shall be ignored and no reply shall be generated.

NOTE This field does not reflect errors that are generated by an encapsulated protocol packet contained within the data portion of the message. For example, an error encountered during an end node's processing of a Set Attributes service.

The status codes shall be as shown in Table 137.

Table 137 – Encapsulation status codes

Status code	Description
0x00	Success
0x01	The sender issued an invalid or unsupported encapsulation command.
0x02	Insufficient memory resources in the receiver to handle the command.
0x03	Poorly formed or incorrect data in the data portion of the encapsulation message.
0x04 – 0x63	Reserved
0x64	An originator used an invalid session handle when sending an encapsulation message to the target.
0x65	The target received a message of invalid length (see 4.3.1.5).
0x66 – 0x68	Reserved
0x69	Unsupported protocol revision.
0x6A – 0xFFFF	Reserved

4.3.1.8 Sender context field

The sender of the command shall assign the value in the Sender Context field of the header. The receiver shall return this value in its reply.

4.3.1.9 Options field

To encapsulate PDUs from the Network and Transport Layer, the Options flags shall be set to zero. The allocation of Option flags shall be as shown in Table 138.

Table 138 – Options flags

Options flags	Definition
Bits 0 – 15	Allocated for compatibility with existing protocols.
Bits 16 – 31	Area for future expansion.

4.3.2 Command descriptions

4.3.2.1 Nop

Either an originator or a target may send a Nop request. No reply shall be generated. The data portion of the request shall be from 0 to 65 511 octets long. The receiver shall ignore any data that is contained in the message.

NOTE A Nop provides a way for either an originator or target to determine if the TCP connection is still open.

The Nop request encapsulation header shall be as shown in Table 139.

Table 139 – Nop request encapsulation header

Field name	Field value
Command	Nop (0)
Length	Length of data portion
Session handle	Ignored
Status	0
Sender context	Ignored
Options	0

4.3.2.2 RegisterSession

An originator shall send a RegisterSession request to a target to initiate a session.

NOTE See 11.7 for detailed information on establishing and maintaining a session.

The RegisterSession request encapsulation header shall be as shown in Table 140.

Table 140 – RegisterSession request encapsulation header

Field name	Field value
Command	RegisterSession (0x65)
Length	4
Session handle	0
Status	0
Sender context	Sender context
Options	0

The parameters in the data portion shall determine the version of the protocol and any session options as shown in Table 141. The protocol version shall be set to 1. The session options shall be set to 0.

Table 141 – RegisterSession request data portion

Field	Type	Description
Protocol version	UINT	Requested protocol version (1)
Options flags	UINT	Session options (0)

Table 142 below shows the organisation of the Options flags:

Table 142 – Options flags

Options flags	Definition
Bits 0 – 7	Allocated for compatibility with existing protocols
Bits 8 – 15	Reserved for future expansion

The target shall send a RegisterSession reply to indicate that it has registered the originator. The reply shall have the same format as the request as shown in Table 143.

Table 143 – RegisterSession reply encapsulation header

Field name	Field value
Command	RegisterSession (0x65)
Length	4
Session handle	Handle returned by target
Status	0
Sender context	Context preserved from request
Options	0

The Session Handle field of the header shall contain a target-generated identifier that the originator shall save and insert in the Session Handle field of the header for all subsequent requests to that target. This field shall be valid only if the Status field is zero (0).

The Sender Context field of the header shall contain the same values present in the original sender request. If the originator has been registered with the target, the Status field shall be zero (0). If the target was unable to register, the Status field shall be non-zero.

The data portion of the reply shall have the same format as the request as shown in Table 144.

Table 144 – RegisterSession reply data portion

Field	Type	Description
Protocol Version	UINT	Protocol version supported
Options	UINT	Session options preserved from request

The Protocol Version field shall equal the requested version if the originator was successfully registered. If the target does not support the requested version of the protocol:

- the session shall not be created;
- the Status field shall be set;
- the target shall return the highest supported version in the Protocol Version field.

If all requested options are supported, the Options field shall return the originator's value. This value shall be zero.

4.3.2.3 UnRegisterSession

Either an originator or a target may send this request to terminate the session. The receiver shall initiate a close of the underlying TCP/IP connection when it receives this request. The session shall also be terminated when the transport connection between the originator and target is terminated. The receiver shall perform any other associated cleanup required on its end. There shall be no reply to this command.

The UnregisterSession request format shall be as shown in Table 145.

Table 145 – UnRegisterSession request encapsulation header

Field name	Field value
Command	UnRegisterSession (0x65)
Length	4
Session Handle	Handle from RegisterSession
Status	0
Sender Context	Sender context
Options	0

The Session Handle shall be set to the value obtained by the original RegisterSession reply. The handle shall become invalid after this request has been received and processed by the target.

4.3.2.4 ListServices

The ListServices request shall be sent to determine which encapsulation service classes the target device supports.

NOTE Each service class has a unique type code, and an optional ASCII name.

One service class is defined, with type code 0x100 and name "Communications". This service class shall indicate that the device supports encapsulation of Type 2 PDU's. All devices that

support Type 2 PDU encapsulation shall support the ListServices command and Communications service class.

The ListServices request encapsulation header shall be as shown in Table 146.

Table 146 – ListServices request encapsulation header

Field name	Field value
Command	ListServices (0x04)
Length	0
Session handle	Ignored
Status	0
Sender context	Chosen by sender
Options	0

The receiver shall reply with an encapsulation message consisting of the header and data, as shown in Table 147 and Table 148. The data portion of the reply shall provide the information on the services supported.

Table 147 – ListServices reply encapsulation header

Field name	Field value
Command	ListServices (0x04)
Length	0
Session handle	Ignored
Status	0
Sender context	Preserved from request
Options	0

The data portion of the reply shall contain a 2-octet object count followed by an array of objects describing the service(s) provided, as shown in Table 148.

Table 148 – ListServices reply data portion

Field name	Type	Description
Object count	UINT	Number of services objects to follow
Type code	UINT	Service type code
Object length	UINT	Remaining length in octets
Version	UINT	Encapsulation protocol version
Flags	UINT	Capability Flags
Name	ARRAY of 16 USINT	Name of service

The Type Code shall identify the service class as shown in Table 149.

Table 149 – Service type codes

Service type code	Description
0x00 – 0xFF	Reserved
0x100	Communications service
0x101 – 0xFFFF	Reserved

The Version field shall indicate the version of the service supported by the target to help maintain compatibility between applications.

Each service shall have a different set of capability flags. Unused flags shall be set to zero.

The Capability Flags, defined for the Communications service, shall be as shown in Table 150.

Table 150 – Communications capability flags

Flag value	Description
Bits 0 – 4	Reserved
Bit 5	If the device supports Type 2 PDU encapsulation this bit shall be set (= 1) ; otherwise, it shall be clear (= 0)
Bits 6 – 7	Reserved
Bit 8	Supports Class 0 and 1 UDP-based connections
Bits 9 – 15	Reserved

The Name field shall allow up to a 16-octet, NULL-terminated ASCII string for descriptive purposes only. The 16-octet limit shall include the NULL character.

4.3.2.5 ListIdentity

A connection originator may use the ListIdentity request to locate and identify potential targets. This request shall be sent as a broadcast message using UDP and does not require that a session be established.

The ListIdentity request encapsulation header shall be as shown in Table 151.

Table 151 – ListIdentity request encapsulation header

Field name	Field value
Command	ListIdentity (0x63)
Length	0
Session handle	This field is ignored since a session need not be established before sending the ListIdentity request
Status	0
Sender context	0
Options	0

One reply item is defined for this command, Target Identity, with item type code 0x0C. This item shall be supported (returned) by all Type 2 capable devices.

Each receiver of the ListIdentity command shall reply with an encapsulation message consisting of the header and data, as shown in Table 152 and Table 153. The data portion of the message shall provide the information on the targets identity. The reply shall be sent to the IP address from which the broadcast request was received.

Table 152 – ListIdentity reply encapsulation header

Field name	Field value
Command	List Identity (0x63)
Length	0
Session handle	Ignored
Status	0
Sender context	0
Options	0

The data portion of the reply is structured as a Common Packet Format that contains a 2-octet object count followed by an array of objects providing the target identity, as shown in Table 153.

At a minimum, the CFP 2 Identity item shall be returned and has the format as defined in Table 153. Part of this item definition follows the Get Attribute All service response definition of the Identity object (data returned based on instance one of this object), and may adopt new members if and when new members are added to that service response. Unlike most fields in the Common Packet Format, the Socket Address field shall be sent in big endian order.

Table 153 – ListIdentity reply data portion

Field name	Type	Description
Object count	UINT	Number of Identity objects to follow
Type code	UINT	Code indicating object type of CPF 2 Identity (0x0C)
Object length	UINT	Remaining length in octets
Version	UINT	Encapsulation protocol version
Socket Address	ARRAY of :	Socket Address
	INT	sin_family (big-endian)
	UINT	sin_port (big-endian)
	UDINT	sin_addr (big-endian)
	ARRAY of USINT	sin_zero (length of 8) (big-endian)
Vendor ID[a]	UINT	Device manufacturers Vendor ID
Device Type[a]	UINT	Device Type of product
Product Code[a]	UINT	Product Code assigned with respect to device type
Revision[a]	USINT[2]	Device revision
Status[a]	WORD	Current status of device
Serial Number[a]	UDINT	Serial number of device
Product Name[a]	SHORT_STRING	Human readable description of device
State[a]	USINT	Current state of device

[a] These parameters are further defined by the corresponding instance attribute of the Identity object.

4.3.2.6 ListInterfaces

The optional ListInterfaces request shall be used by a connection originator to identify potential non-Type 2 communication interfaces associated with the target. A session need not be established to send this command.

The ListInterfaces request encapsulation header shall be as shown in Table 154.

Table 154 – ListInterfaces request encapsulation header

Field name	Field value
Command	List Interfaces (0x64)
Length	0
Session handle	Ignored
Status	0
Sender context	0
Options	0

If supported, the receiver of a ListInterfaces request command shall reply an encapsulation message consisting of the header and data, as shown below in Table 155.

Table 155 – ListInterfaces reply encapsulation header

Field name	Field value
Command	List Interfaces (0x64)
Length	0
Session handle	Ignored
Status	0
Sender context	0
Options	0

The data portion of the message is structured as a Common Packet Format, which contains a 2-octet object count followed by an array of objects providing information on the non-Type 2 communication interfaces associated with the target.

There are no publicly defined items returned with this reply. The vendor-specific item(s) which is/are returned shall, at a minimum, return a 32 bit Interface Handle which is used by other encapsulation commands, for example, the SendRRData command.

4.3.2.7 SendRRData

A SendRRData request shall transfer an encapsulated request/reply packet between the originator and target, where the originator initiates the command. The actual request/reply packets shall be encapsulated in the data portion of the message and shall be the responsibility of the target and originator. The SendRRData command shall be used to send encapsulated UCMM messages.

The SendRRData request encapsulation header shall be as shown in Table 156.

Table 156 – SendRRData request encapsulation header

Field name	Field value
Command	SendRRData (0x6F)
Length	Length of data portion
Session handle	Handle returned by RegisterSession
Status	0
Sender context	Sender context
Options	0

The data portion of the message shall contain request parameters and the encapsulated protocol packet as shown in Table 157.

Table 157 – SendRRData request data portion

Field name	Type	Description
Interface handle	UDINT	Shall be 0
Timeout	UINT	Operation Timeout
Encapsulated protocol packet (see Common Packet Format specification in 4.3.3)		

The Interface handle shall identify the Communications Interface to which the request is directed. This handle shall be 0 for encapsulating Type 2 PDUs.

The target shall abort the requested operation after the timeout expires. Timeout values shall be expressed in seconds. The minimum timeout value shall be one (1) second; the maximum value shall be 65 535 s (about 18 h). A value of zero shall indicate that the operation shall only be aborted when the encapsulated protocol times-out. Since the Network and Transport Layer already defines a timeout mechanisms for both connected and unconnected messages, the Timeout field shall be zero.

The encapsulated protocol packet shall be encoded in a Common Packet Format as shown in 4.3.3.

The SendRRData reply, as shown in Table 158, shall contain data in response to the SendRRData request. The reply to the original encapsulated protocol request shall be contained in the data portion of the SendRRData reply.

Table 158 – SendRRData reply encapsulation header

Field name	Field value
Command	SendRRData (0x6F)
Length	Length of data portion
Session handle	Handle returned by RegisterSession
Status	0
Sender context	Preserved sender context
Options	0

The format of the data portion of the reply message shall be the same as that of the SendRRData request message.

4.3.2.8 SendUnitData

The SendUnitData request shall send encapsulated connected messages. A reply need not be returned. This command may be used when the encapsulated protocol has its own underlying end-to-end transport mechanism.

The format of the SendUnitData request shall be as shown in Table 159.

Table 159 – SendUnitData request encapsulation header

Field name	Field value
Command	SendUnitData (0x70)
Length	Length of data portion
Session handle	Handle returned by RegisterSession
Status	0
Sender context	Sender context
Options	0

The data portion of the message shall contain request parameters as well as the encapsulated protocol packet as shown in Table 160.

Table 160 – SendUnitData request data portion

Field name	Type	Description
Interface handle	UDINT	Shall be 0
Timeout	UINT	Operation timeout
Encapsulated protocol packet (see Common Packet Format specification in 4.3.3).		

Interface handle and Timeout shall be the same as in the SendRRData request.

4.3.3 Common packet format

4.3.3.1 General

The common packet format shall consist of an item count, followed by an address item, then a data item (in that order) as shown in Table 161. Additional optional items may follow.

NOTE The common packet format defines a standard format for protocol packets that are transported with the encapsulation protocol. The common packet format is a general-purpose mechanism designed to accommodate future packet or address types.

Table 161 – Common packet format

Field name	Type	Description
Item count	UINT	Number of items to follow (shall be at least 2)
Address item	Item Struct (see below)	Addressing information for encapsulated packet
Data item	Item Struct (see below)	The encapsulated data packet
Optional additional items.		

The address and data item structure shall be as shown in Table 162.

Table 162 – Address and data item structure

Field name	Type	Description
Type ID	UINT	Type of item encapsulated
Length	UINT	Length in octets of data to follow
Data	Variable	The data (if length >0)

The address type ID's shall be as shown in Table 163.

Table 163 – Address type ID's

Address type ID	Description
0x00	Null (used for UCMM messages)
0x01 – 0xA0	Reserved
0xA1	Connection-based (used for connected messages)
0xA2 – 0x8001	Reserved
0x8002	Sequenced Address Type
0x8003 – 0xFFFF	Reserved

The data type ID's shall be as shown in Table 164.

Table 164 – Data type ID's

Data type ID	Description
0x00 – 0xB0	Reserved
0xB1	Connected Transport PDU
0xB2	Unconnected message
0xB3 – 0x7FFF	Reserved
0x8000	Sockaddr Info, originator-to-target
0x8001	Sockaddr Info, target-to-originator
0x8002 – 0xFFFF	Reserved

4.3.3.2 Address types

4.3.3.2.1 Null

The null address type shall contain only the type id and the length as shown in Table 165. The length shall be zero. No data shall follow the length. Since the null address type contains no routing information, it shall be used when the protocol packet itself contains any necessary routing information. The null type shall be used for Unconnected Messages.

Table 165 – Null address type

Field name	Type	Field value
Type ID	UINT	0
Length	UINT	0

4.3.3.2.2 Connected

This address type shall be used when the encapsulated protocol is connection-oriented. The data shall contain a connection identifier, as shown in Table 166.

NOTE Connection identifiers are exchanged in the Forward_Open service of the Connection Manager.

Table 166 – Connected address type

Field name	Type	Field value
Type ID	UINT	0xA1
Length	UINT	4
Data	UDINT	Connection Identifier

4.3.3.2.3 Sequenced address type

This address type shall be used for class 0 and class 1 connected data. The data shall contain a connection identifier and a sequence number, as shown in Table 167. Usage is further described in 11.3.3.

Table 167 – Sequenced address type

Field name	Type	Field value
Type ID	UINT	0x8002
Length	UINT	8
Data	UDINT	Connection Identifier
	UDINT	Sequence Number

4.3.3.3 Data types

4.3.3.3.1 Unconnected

The data type that encapsulates an unconnected message shall be as shown in Table 168.

Table 168 – UCMM data type

Field name	Type	Field value
Type ID	UINT	0xB2
Length	UINT	Length, in octets, of the unconnected message
Data	Variable	The unconnected message

The format of the unconnected message shall be the same as PDU destined for the Message Router. The UCMM header, defined in 4.1.3, shall not be present. The context field in the encapsulation header shall be used for unconnected request/reply matching.

4.3.3.3.2 Connected

The data type that encapsulates a connected transport PDU shall be as shown in Table 169.

Table 169 – Connected data type

Field name	Type	Field value
Type ID	UINT	0xB1
Length	UINT	Length, in octets, of the transport PDU
Data	Variable	The transport PDU

4.3.3.3.3 Sockaddr info

The Sockaddr Info items shall be used to encapsulate socket address information necessary to send datagrams (the connected data) between the target and originator. There are separate items for originator-to-target and target-to-originator socket information.

The Sockaddr Info items shall have the structure shown in Table 170.

Table 170 – Sockaddr info items

Field name	Type	Field value
Type ID	UINT	0x8000, 0x8001
Length	UINT	16 (octets)
Data	ARRAY	See below

The data portion of the Sockaddr Info type shall follow the format of the sockaddr_in structure as defined in the Winsock specification, version 1.1. The sockaddr_in structure is shown below:

```
struct sockaddr_in {
            short    sin_family;
            u_short  sin_port;
            struct   in_addr sin_addr;
            char     sin_zero[8];
};
```

The multi-octet fields in the sockaddr_in structure shall be returned in TCP/IP octet order, since this is the octet ordering used by the Winsock API.

5 Transfer syntax

5.1 Compact encoding

5.1.1 Encoding rules

This subclause describes the means by which the data types defined in this standard shall be encoded/transferred across the control network. The abstract syntax definition along with a particular set of encoding rules shall result in the transfer syntax. For application user data, a single set of encoding rules shall be defined (Compact Encoding), resulting in the Compact transfer syntax.

Compact Encoding rules shall start with the encoding rules defined in ISO/IEC 8825. Compact Encoding then applies optimisation rules, starting with the outer most Service Data Unit (SDU) and progressing to each successive encapsulated SDU. Compact Encoding shall define a more efficient encoding mechanism by reducing the amount of information (overhead) transferred between devices.

The difference between a Compact encoded value and an ASN.1 encoded value shall be the removal of the fields describing the type and length of the information. The TAG and LENGTH components of an ASN.1-encoded value shall not be transmitted on the control network. In addition, the Compact Encoding rules shall indicate that octet ordering rules are the reverse of those seen in ASN.1.

Given the conditions listed in 5.1.2, general rules shall be applied to an ASN.1 encoded value to generate a Compact encoded value. The general rules shall be as follows:

- remove the Identifier Octets;

- remove the “TAG” octets specified by ASN.1;
- remove the Length Octets;
- remove the “LENGTH” octets specified by ASN.1;
- reverse the octet ordering for multiple content octets.

5.1.2 Encoding constraints

The representation of a variable value using Compact Encoding shall be possible with the following restrictions:

- the variable type shall be fixed length and shall have no conditional or optional fields;
- the encoding of a given variable shall be represented with a constant number of octets derived from the type specification of this variable.

5.1.3 Examples

5.1.3.1 BOOL encoding

The BOOLEAN encoding shall be performed on a single OCTET as described in Table 171.

Table 171 – BOOLEAN encoding

If the value is:	Then:
FALSE	bit 0 of the octet is 0 (’00’H)
TRUE	bit 0 of the octet is 1 (’01’H)

A FALSE BOOL shall be represented as shown in Table 172.

Table 172 – Example compact encoding of a BOOL value

Octet number	1st
BOOL	00

5.1.3.2 SignedInteger encoding

The SignedInteger encoding shall be performed as described in Table 173.

Table 173 – Encoding of SignedInteger values

Octet number	1st	2nd	3rd	4th	5th	6th	7th	8th
SINT	0LSB							
INT	0LSB	1LSB						
DINT	0LSB	1LSB	2LSB	3LSB				
LINT	0LSB	1LSB	2LSB	3LSB	4LSB	5LSB	6LSB	7LSB
NOTE The example in Table 174 illustrates the encoding of a variable of type DINT whose value is 0x12345678.								

Table 174 – Example compact encoding of a SignedInteger value

Octet number	1st	2nd	3rd	4th
DINT	78	56	34	12

5.1.3.3 UnsignedInteger encoding

The UnsignedInteger encoding shall be performed as described in Table 175.

Table 175 – UnsignedInteger values

Octet number	1st	2nd	3rd	4th	5th	6th	7th	8th
USINT	0LSB							
UINT	0LSB	1LSB						
UDINT	0LSB	1LSB	2LSB	3LSB				
ULINT	0LSB	1LSB	2LSB	3LSB	4LSB	5LSB	6LSB	7LSB
NOTE Table 176 illustrates the encoding of a variable of type UDINT whose value is 0xAABBCCDD.								

Table 176 – Example compact encoding of an UnsignedInteger

Octet number	1st	2nd	3rd	4th
UDINT	DD	CC	BB	AA

5.1.3.4 FixedLengthReal encoding

The FixedLengthReal encoding shall be performed as described in Table 177.

Table 177 – FixedLengthReal values

Octet number	1st	2nd	3rd	4th	5th	6th	7th	8th
REAL	0LSB	1LSB	2LSB	3LSB				
LREAL	0LSB	1LSB	2LSB	3LSB	4LSB	5LSB	6LSB	7LSB

Table 178 illustrates the encoding of a variable (Float1) whose type is REAL and whose value is Float1: = 10,0.

NOTE 1 The assignment of the value is using the IEC 61131-3 notation. The ASN.1 value is {'41200000'H} in IEEE format : 1,25*23 , exponent is 130 (bias 127), fraction is 25.

Table 178 – Example compact encoding of a REAL value

Octet contents	0LSB	1LSB	2LSB	3LSB
REAL	00	00	20	41

Table 179 illustrates the encoding of a variable (Float2) whose type is LREAL and whose value is Float2: = –100,0.

NOTE 2 The ASN.1 value is {'C059000000000000'H} in IEEE format : 1,562 5*26, exponent is 1 029 (bias 1 023), fraction is 0,562 5.

Table 179 – Example compact encoding of a LREAL value

Octet contents	0LSB	1LSB	2LSB	3LSB	4LSB	5LSB	6LSB	7LSB
LREAL	00	00	00	00	00	00	59	C0

5.1.3.5 Time encodings

Table 180 illustrates the encoding of Real numbers with fixedlengths.

Table 180 – FixedLengthReal values

Octet number	1st	2nd	3rd	4th	5th	6th	7th	8th
TIME	0LSB	1LSB	2LSB	3LSB				
DATE	0LSB	1LSB						
TIME_OF_DAY	0LSB	1LSB	2LSB	3LSB				
DATE_AND_TIME	0LSB-Time	1LSB-Time	2LSB-Time	3LSB-Time	0LSB-Date	1LSB-Date		
FTIME	0LSB	1LSB	2LSB	3LSB				
LTIME	0LSB	1LSB	2LSB	3LSB	4LSB	5LSB	6LSB	7LSB

5.1.3.6 String encodings

This subclause gives examples of the Compact Encoding of STRING, STRING2, STRINGN, and SHORT_STRING data values.

NOTE The preferred string type for user supplied string data is STRING2 due to international character string requirements.

Table 181 provides a generic illustration of the encoding of a STRING value.

Table 181 – STRING value

	Contents (charcount)		Contents (string contents)
STRING	0LSB	1LSB	0LSB

1 octet characters

Table 182 provides a generic illustration of the encoding of a STRING2 value.

Table 182 – STRING2 value

	Contents (charcount)		Contents (string2contents)	
STRING2	0LSB	1LSB	0LSB	1LSB

2 octet characters

Table 183 provides a generic illustration of the encoding of a STRINGN value.

Table 183 – STRINGN value

	Contents (charsize)		Contents (charcount)		Contents (stringNcontents)	
STRINGN	0LSB	1LSB	0LSB	1LSB	0LSB	NLSB

N octet characters

Table 184 provides a generic illustration of the encoding of a SHORT_STRING value.

Table 184 – SHORT_STRING value

	Contents (charcount)	Contents (short_string)
SHORT_STRING	0LSB	0LSB

1 octet characters

Table 185 illustrates the encoding of a string variable whose contents equal ”Mill”. Encoding examples of all string types are presented. Character coding is specified in ISO/IEC 10646. The hexadecimal equivalent is: {’4D696C6C’H} for 8 bit encoding.

Table 185 encodes ”Mill” as a STRING type.

Table 185 – Example compact encoding of a STRING value

	Contents (charcount)		Contents (string contents)			
STRING	04	00	4D	69	6C	6C

Table 186 encodes ”Mill” as a STRING2 type.

Table 186 – Example compact encoding of STRING2 value

	Contents (charcount)		Contents (string2 contents)							
STRING2	04	00	4D	00	69	00	6C	00	6C	00

Table 187 encodes ”Mill” as a SHORT_STRING type.

Table 187 – SHORT_STRING type

	Contents (charcount)	Contents (short_string contents)			
SHORT_STRING	04	4D	69	6C	6C

5.1.3.7 FixedLengthBitString encoding

This subclause provides examples of the Compact Encoding of SWORD, WORD, DWORD, LWORD data values. Figure 14 illustrates the bit placement rules associated with the Compact Encoding of a FixedLengthBitString.

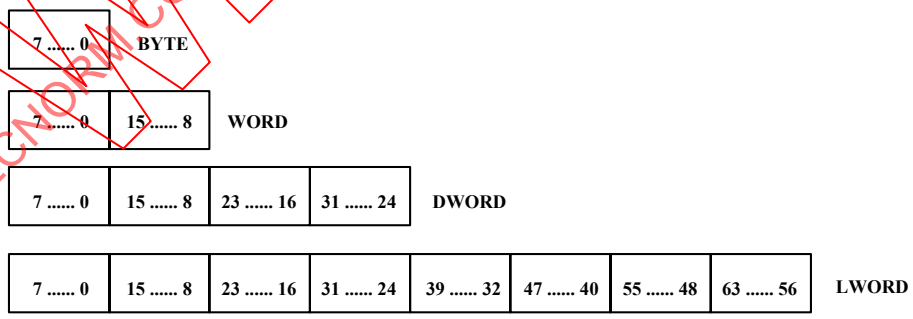


Figure 14 – FixedLengthBitString compact encoding bit placement rules

Figure 15 through Figure 18 illustrate the encoding of SWORD, WORD, DWORD, and LWORD.

Bits In Memory:
7 0
00001111

Compact Encoded BYTE
00001111 or 0x0F

Figure 15 – Example compact encoding of a SWORD FixedLengthBitString

Bits In Memory:
15 0
00001111 11001111

Compact Encoded WORD
11001111 00001111 or 0xCF0F

Figure 16 – Example compact encoding of a WORD FixedLengthBitString

Bits In Memory:
31 0
00001111 11001111 10110110 00111110

Compact Encoded DWORD
00111110 10110110 11001111 00001111 or 0x3EB6CF0F

Figure 17 – Example compact encoding of a DWORD FixedLengthBitString

Bits In Memory:
63 0
11110000 11001111 10110110 00111110 11110000 00101101 00011110 00001111

Compact Encoded LWORD
00001111 00011110 00101101 11110000 00111110 10110110 11001111 11110000 or 0x0F1E2DF03EB6CFF0

Figure 18 – Example compact encoding of a LWORD FixedLengthBitString

5.1.3.8 Array encoding

5.1.3.8.1 One-dimensional array encoding

The Array encoding shall use the encoding rules for the data types for each element and concatenates the elements which compose the array. The encoded values of the array elements shall be encoded in the same order as they are declared in the corresponding ASN.1 type or variable specification. These elements may be of any data type.

The ASN.1–style definition of a single–dimensional array in the control network shall be:

```
ARRAY ::= SEQUENCE OF { array_dimension_low_bound,
            array_dimension_high_bound, NetworkData }

Assume the following array definition::
ARRAY1 ::= SEQUENCE OF {array_dimension_low_bound := 0,
           array_dimension_high_bound := 1, UINT}
```

Plugging the UINT values {1,2} into this array definition yields the encoding specified in Table 188.

Table 188 – Example compact encoding of a single dimensional ARRAY

Octet number	1st	2nd	3rd	4th
ARRAY	01	00	02	00

5.1.3.8.2 Two-dimensional array encoding

```
ARRAY ::= SEQUENCE OF { array_dimension_low_bound,
array_dimension_high_bound,
      SEQUENCE OF { array_dimension_low_bound,
array_dimension_high_bound, NetworkData } }
```

5.1.3.8.3 Three-dimensional array encoding

```
ARRAY ::= SEQUENCE OF { array_dimension_low_bound,
   array_dimension_high_bound,
   SEQUENCE OF { array_dimension_low_bound,
   array_dimension_high_bound,
   SEQUENCE OF { array_dimension_low_bound,
    array_dimension_high_bound, NetworkData } } }
```

Since control network data may comprise either ElementaryData or DerivedData, a new type or variable specification may be required before transmitting the values for the ARRAY.

A multi–dimensional array shall be seen on the wire as a single–dimensional array. The order of the array elements shall be maintained via the packing/unpacking sequence followed by the end nodes. The sequence followed shall be to access the Nth dimension of the array for all values of the other dimensions.

This shall be achieved by first accessing the array with all dimensions set to their initial index values. After this the Nth dimension is incremented through all of its index values. When the end of the index range for the Nth dimension is reached, the (N-1)th dimension is incremented, and the Nth dimension is set to its initial index value. This process is repeated until all of the array's dimensions have reached the ends of their index ranges, and results in the array being packed into the message buffer as a single–dimensional array. The same procedure is followed to unpack the single–dimensional array into a multi–dimensional array.

The two–dimensional array

```
ARRAY1 [0..1 , 0..2] of UINT := { { 1, 2, 3 },
                                  { 4, 5, 6 } }
```

results in the data stream shown in Table 189 when it is packed into a single–dimensional array following the compact encoding rules:

Table 189 – Example compact encoding of a multi-dimensional ARRAY

Octet number	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th	12th
ARRAY	01	00	02	00	03	00	04	00	05	00	06	00

5.1.3.9 Structure encoding

The structure encoding shall use the encoding rules for the data types for each element and concatenates the elements which compose the structure.

The encoded values of the structure elements shall be encoded in the same order as they are declared in the corresponding ASN.1 type or variable specification. These elements may be of any Data type.

```
STRUCT ::= SEQUENCE OF NetworkData –- May be different types
```

Since NetworkData may be comprised of either ElementaryData or DerivedData, a new type or variable specification may be required before transmitting the values for the STRUCT.

Assume the following structure definition:

newStruct ::= SEQUENCE { BOOL,UINT,DINT}

Plugging the values {TRUE,'1234'H,'56789ABC'H} into the structure results in the encoding as specified in Table 190.

Table 190 – Example compact encoding of a STRUCTURE

Octet number	1st	2nd	3rd	4th	5th	6th	7th
newStruct	01	34	12	BC	9A	78	56

5.1.3.10 Complex encoded data format examples

5.1.3.10.1 General

The examples 5.1.3.10.2 and 5.1.3.10.3 show how more complex data formats shall be packed. Example 5.1.3.10.2 shows the packing of an array of structures. Example 5.1.3.10.3 shows how a structure with an array element is packed.

5.1.3.10.2 Example 1: encoding an array of structures:

```
STRUCT1 ::= SEQUENCE OF {
              UINT     ele1;
              USINT    ele2;
              USINT    ele3;
              USINT    ele4;
              UINT     ele5
}
  ARRAY1 [ 0..1 , 0..2 ] of STRUCT1 := {
              { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 },
              { 11, 12, 13, 14, 15 } },
              { { 15, 14, 13, 12, 11 }, { 10, 9, 8, 7, 6 },
              { 5, 4, 3, 2, 1 } } }
```

results in the following data stream:

```
[01] [00] [02] [03] [04] [05] [00]  [06] [00] [07] [08] [09] [0A] [00]
[0B] [00] [0C] [0D] [0E] [0F] [00]  [0F] [00] [0E] [0D] [0C] [0B] [00]
[0A] [00] [09] [08] [07] [06] [00]  [05] [00] [04] [03] [02] [01] [00]
```

5.1.3.10.3 Example 2: encoding a structure with an array element

```
STRUCT2 ::= SEQUENCE OF {
              UINT     ele1;
              ARRAY [ 0..2 ] of USINT array2;
              UINT     ele5;
}
STRUCT2 := { 1, { 2, 3, 4 }, 5 }
```

results in the following data stream:

```
[01] [00]  [02] [03] [04]  [05] [00]
```

5.2 Data type reporting

5.2.1 Object data representation

Objects may choose to implement a mechanism for reporting Data Type of a particular Attribute or transmitting type information along with the actual data. This subclause defines the means by which Data Typing information is conveyed.

The specification of Data Type information utilises the ASN.1 methodology specified in ISO/IEC 8824 and ISO/IEC 8825 with the control network defined optimisations to encode the **DataTypeSpecification** production defined in 4.2.4.

The control network defined optimisation is that the Length Octet of a NULL type is not encoded.

NOTE For example, the encoding of 'abc [PRIVATE 1] IMPLICIT NULL' would be: 0xC1 (a tag with no length octet).

5.2.2 Elementary data type reporting

Elementary data types shall be identified using the identification codes defined in Table 191. These codes shall define the encoding of the primitive members of the DataTypeSpecification production. The control network specifies that ASN.1 NULL types shall not report the Length Octet of zero (0).

Table 191 – Identification codes and descriptions of elementary data types

Data type name	Data type code (in hex)	Data type description
BOOL	C1	Logical Boolean with values TRUE and FALSE
SINT	C2	Signed 8–bit integer value
INT	C3	Signed 16–bit integer value
DINT	C4	Signed 32–bit integer value
LINT	C5	Signed 64–bit integer value
USINT	C6	Unsigned 8–bit integer value
UINT	C7	Unsigned 16–bit integer value
UDINT	C8	Unsigned 32–bit integer value
ULINT	C9	Unsigned 64–bit integer value
REAL	CA	32–bit floating point value
LREAL	CB	64–bit floating point value
STIME	CC	Synchronous time information
DATE	CD	Date information
TIME_OF_DAY	CE	Time of day
DATE_AND_TIME	CF	Date and time of day
STRING	D0	character string (1 octet per character)
SWORD	D1	bit string - 8-bits
WORD	D2	bit string - 16-bits
DWORD	D3	bit string - 32-bits
LWORD	D4	bit string - 64-bits
STRING2	D5	character string (2 octets per character)
FTIME	D6	Duration (high resolution)
LTIME	D7	Duration (long)

Data type name	Data type code (in hex)	Data type description
ITIME	D8	Duration (short)
STRINGN	D9	character string (N octets per character)
SHORT_STRING	DA	character sting (1 octet per character, 1 octet length indicator)
TIME	DB	Duration (milliseconds)
EPATH	DC	Path segments
STRINGI	DE	International Character String

5.2.3 Constructed data type reporting

5.2.3.1 Structure type definition

5.2.3.1.1 General

The control network defines two different methods for reporting Structure type definitions:

- Formal Encoding (FormalStrucTypeSpec);
- Abbreviated Encoding (AbbreviatedStrucTypeSpec).

Formal encoding shall be used to provide a detailed report of the complete structure definition, including the complete definition of all component data types. Abbreviated encoding shall be used to specify a *shorter* form of the structure definition. This *shorter* form shall not include the data types associated with the structure's components.

5.2.3.1.2 Formal encoding for structure type information

The examples 5.2.3.1.3 and 5.2.3.1.4 illustrate formal encoding for structure type specifications. This is actually an example of the encoding of the FormalStrucTypeSpec production defined in 4.2.4.

5.2.3.1.3 Example 1

Table 192 illustrates the encoding of the following structure definition.

```
STRUCT ::= SEQUENCE OF { BOOL, UINT, DINT }
```

Table 192 – Example 1 of formal encoding of a structure type specification

STRUCT type	Type length	Component types	Component types	Component types
		BOOL	UINT	DINT
A2	03	C1	C7	C4
NOTE The IMPLICIT NULL types from the DataTypeSpecification production are not followed by a Length Octet of zero (0).				

5.2.3.1.4 Example 2

Figure 19 illustrates the encoding of the following structure definition.

```
STRUCT_MAIN ::= SEQUENCE OF { UINT, STRUCT_SUB, INT }

with subelement STRUCT_SUB defined as:

STRUCT_SUB ::= SEQUENCE OF { UINT, SINT, INT }
```

Struct Type	Type Length	Component Types						
		UINT	Struct Type	Type Length	Nested Structure Component Types			
					UINT	SINT	INT	INT
A2	07	C7	A2	03	C7	C2	C3	C3

Figure 19 – Example 2 of formal encoding of a structure type specification

5.2.3.1.5 Abbreviated encoding for structure type information

The example 5.2.3.1.6 illustrates the abbreviated encoding for structure type specifications. This is actually an example of the encoding of the AbbreviatedStrucTypeSpec production defined in 4.2.4.

The UINT defined within the AbbreviatedStrucTypeSpec production shall be initialised with a 16 bit Cyclic Redundancy Check (CRC) value (see IEC 61158-4-2). This can be used by application logic to determine whether or not the format of the structure has changed. The octet stream used to produce the CRC is the actual formally encoded (FormalStrucTypeSpec) structure type specification.

5.2.3.1.6 Example

Figure 20 shows the abbreviated encoding of the structure definition presented in 5.2.3.1.5:

Struct Type	Type Length	UINT containing CRC	
A0	02	C7	26

Figure 20 – Example of abbreviated encoding of a structure type specification

NOTE The algorithms presented in this STANDARD are used to generate the CRC value of 0x26C7 from the Formally Encoded type specification: [A2][07][C7][A2][03] [C7][C2][C3][C3].

5.2.3.2 Array type definition

Two different methods for reporting array type definitions shall be:

- Formal Encoding (FormalArrayTypeSpec);
- Abbreviated Encoding (AbbreviatedArrayTypeSpec).

Formal encoding shall be used to provide a detailed report of the complete array definition, including the data content and the array's dimensions. Abbreviated encoding shall be used to specify a *shorter* form of the array definition. This *shorter* form shall not include information specifying the array's dimensions.

5.2.3.3 Formal encoding for array type information

5.2.3.3.1 Common notes

NOTE 1 This subclause contains no normative requirements.

NOTE 2 The examples 5.2.3.3.2 and 5.2.3.3.3 illustrate formal encoding for structure type specifications. This is actually an example of the encoding of the FormalArrayTypeSpec production defined in 4.2.4.

5.2.3.3.2 Example 1

Figure 21 shows the formal encoding of the following array definition

```
ARRAY1 ::= SEQUENCE OF { array_dimension_low_bound := 0,
                         array_dimension_high_bound := 9,
                         UINT
```

Array Type	Type Length	Lower Bound Tag	Lower Bound Length	Lower Bound	Upper Bound Tag	Upper Bound Length	Upper Bound	UINT
A3	07	80	01	00	81	01	09	C7

Figure 21 – Example 1 of formal encoding of an array type specification

NOTE The IMPLICIT NULL types from the DataTypeSpecification production are not followed by a Length Octet of zero (0).

5.2.3.3.3 Example 2

Figure 22 shows the encoding of the following array definition.

```
ARRAY1 ::= SEQUENCE OF { array_dimension_low_bound := 0,
                         array_dimension_high_bound := 19,
                       SEQUENCE OF { array_dimension_low_bound := 0,
                                     array_dimension_high_bound := 255,
STRUCT_ELE } }
```

in which `STRUCT_ELE` is defined as:

```
STRUCT_ELE ::= SEQUENCE OF { UINT, SINT, INT }
```

Formal Encoding: [A3][13][80][01][00][81][01][13][A3][0B][80][01][00][81][01][FF][A2][03][C7][C2][C3]

Description:

Array Type	Type Length	Lower Bound Tag	Lower Bound Length	Lower Bound	Upper Bound Tag	Upper Bound Length	Upper Bound
A3	13	80	01	00	81	01	13

Array Contents

Array Type	Type Length	Lower Bound Tag	Lower Bound Length	Lower Bound	Upper Bound Tag	Upper Bound Length	Upper Bound
A3	0B	80	01	00	81	01	FF

Nested Array Contents

Struc Type	Type Length	UINT	SINT	INT
A2	03	C7	C2	C3

Figure 22 – Example 2 of formal encoding of an array type specification

NOTE The IMPLICIT NULL types from the DataTypeSpecification production are not followed by a Length Octet of zero (0).

5.2.3.4 Abbreviated tag encoding for array type information

5.2.3.4.1 Common notes

NOTE 1 This subclause contains no normative references.

NOTE 2 The abbreviated encoding of an Array type shall not include the information specifying the Array's dimensions. This is actually an example of the encoding of the AbbreviatedArrayTypeSpec production defined in 4.2.4.

5.2.3.4.2 Example 1

Figure 23 shows the abbreviated encoding of the following array definition:

```
ARRAY2 ::= SEQUENCE OF { array_dimension_low_bound := 0,
                         array_dimension_high_bound := 9,
                         UINT }
```

Array Type	Type Length	UINT
A1	01	C7

Figure 23 – Example 1 of abbreviated encoding of an array type specification

NOTE The IMPLICIT NULL types from the DataTypeSpecification production are not followed by a Length Octet of zero (0).

5.2.3.4.3 Example 2

Figure 24 shows the abbreviated encoding of the following array definition:

```
ARRAY ::= SEQUENCE OF { array_dimension_low_bound := 0,
                        array_dimension_high_bound := 19,
                        SEQUENCE OF { array_dimension_low_bound := 0,
                        array_dimension_high_bound := 899,
STRUCT_ELE } }
```

in which `STRUCT_ELE` is defined as:

```
STRUCT_ELE ::= SEQUENCE OF { UINT, SINT, INT }
```

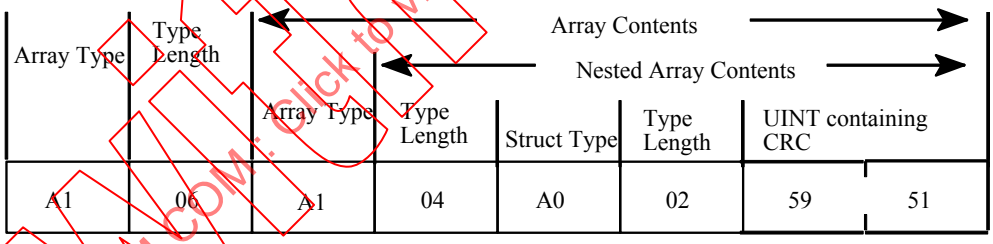


Figure 24 – Example 2 of abbreviated encoding of an array type specification

6 Structure of FAL protocol state machines

Interface to FAL services and protocol machines are specified in this subclause. Conventions used for the descriptions are given in the generic part of this IEC Standard.

This fieldbus follows the structure outlined for Type 1 fieldbus with the following specific features :

1. There is no formal definition of AP-Context Machine

2. There is a formally-defined FSPM Machine serving as an interface between FAL User and ARPM.

3. There are ARPM Machines of two different types :

   a) one ARPM machine for connection-less application relationships

   b) seven ARPM machines for connection-oriented application relationships

4. DMPM Machine is defined at the interface to the Type 2 Data-link layer

7 AP-Context state machine

7.1 Overview

Type 2 supports the AP-Context State Machine specified in, when the Connection object is also supported.

7.2 Connection object state machine

7.2.1 I/O Connection instance behavior

Figure 25 provides a general overview of the behavior associated with an **I/O Connection object** (**instance_type** attribute = I/O).

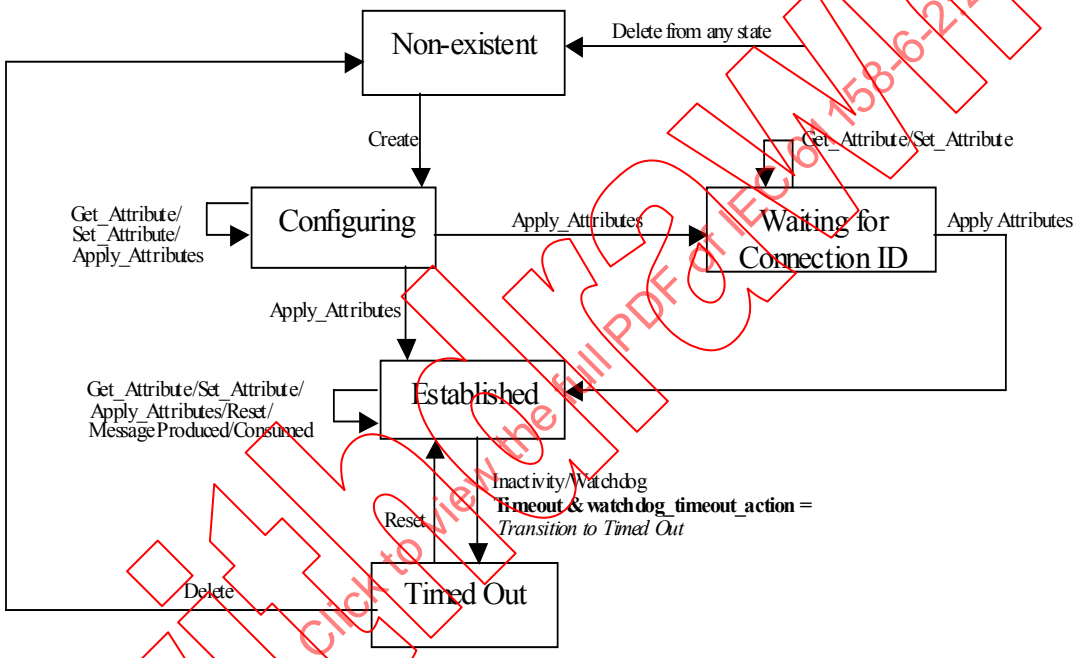


Figure 25 – I/O Connection object state transition diagram

Important: Table 193 provides a detailed State Event Matrix for an I/O Connection object and implementations should be based on this information. This State Event Matrix does not dictate rules with regards to product specific, internal logic. Any attempt to access the Connection Class or a Connection object Instance may need to pass through product specific verification. This may result in an error scenario that is not indicated by the SEM in Table 193. This may also result in additional, product specific indications delivered from a Connection object to the application and/or a specific application object. The point to remember is that the Connection object shall exhibit the externally visible behavior specified by the SEM and the attribute definitions.

Table 193 – I/O Connection state event matrix

Event	I/O Connection object state				
	Non-Existent	**Configuring**	**Waiting for Connection ID**	**Established**	**Timed Out**
Connection Class receives a Create Request	Class instantiates a Connection object. Set instance_type to I/O. Set all other attributes to default values. Transition to Configuring	Not applicable	Not applicable	Not applicable	Not applicable
Connection Class receives a Delete Request	Error: Object does not exist (General Error Code 16_{hex})	Release all associated resources. Transition to Non-existent	Release all associated resources. Transition to Non-existent.	Release all associated resources. Transition to Non-existent.	Release all associated resources. Transition to Non-existent.
Set_Attribute _Single	Error: Object does not exist (General Error Code 16_{hex})	Validate/service the request based on internal logic and per the Access Return response.	If request to modify produced or consumed_connection_ id then validate the value and service the request. Return appropriate response. If this is a request to access an attribute other than the produced or consumed_connection_ id, then return an Error Response whose General Error Code is set to $0C_{hex}$ (The object can not perform the requested service in its current mode/state)	Validate/service the request based on internal logic and per the Access Rules Return appropriate response.	Validate/service the request based on internal logic and per the Access Rules Return appropriate response.
Get_Attribute _Single	Error: Object does not exist (General Error Code 16_{hex})	Validate/service the request based on internal logic and per the Access Rules Return response.	Validate/service the request based on internal logic and per the Access Rules Return response.	Validate/service the request based on internal logic and per the Access Rules Return response.	Validate/service the request based on internal logic and per the Access Rules Return response.
Reset	Error: Object does not exist (General Error Code 16_{hex})	Error: The object cannot perform the requested service in its current mode/state. (General Error Code value =)	Error: The object cannot perform the requested service in its current mode/state. (General Error Code value = $0C_{hex}$)	Cancel the current Inactivity/Watchdog Timer. Using the value in the expected_packet_rate attribute, re-start the Inactivity/Watchdog Timer. A success response is returned even if an Inactivity/Watchdog Timer is not utilized by the Connection object (Client Transport Class 0, expected_packet_rate = $00C_{hex}$).	Using the value in the expected_packet_rate attribute, start the Inactivity/Watchdog timer and transition back to the Established state. If the expected_packet_rate attribute has been set to zero (0) while the Connection was in the Timed Out state, then just transition back to Established without activating an Inactivity/Watchdog Timer.

Event	I/O Connection object state				
	Non-Existent	**Configuring**	**Waiting for Connection ID**	**Established**	**Timed Out**
Apply_ Attributes	Error: Object does not exist (General Error Code 16_{hex})	Deliver Connection object to the application which validates the attribute information. If either of the connection_id attributes (produced or consumed) needs to be configured and cannot be generated by this module, then perform all the other steps necessary to configure the connection, return a Successful Response and transition to the Waiting For Connection ID state. The inability to generate a produced or consumed connection ID value IS NOT reported as an error. If all attributes are validly configured, then perform all the steps necessary to satisfy this connection, start all required timers, and transition to the Established state. If an error is detected, then an Error Response is returned and the Connection remains in the Configuring state[a] and, if a Client Connection with a production trigger value of 0 or 1 (Cyclic or Change of State), produce initial data.	If either of the connection_id attributes (produced or consumed) still needs to be configured and cannot be generated by this module, then return a Successful Response and remain in the Waiting For Connection ID state. The inability to generate a produced or consumed connection ID value IS NOT reported as an error. If the produced and/or consumed_connection_ id attributes are now validly configured, then transition to the Established state and return a Successful Response[1] and, if a Client Connection with a production trigger value of 0 or 1 (Cyclic or Change of State), produce initial data.	All modifications take place immediately once the Connection has transitioned to the Established state. Return Error: The object cannot perform the requested service in its current mode/state. (General Error Code value = $0C_{hex}$)	Error: The object cannot perform the requested service in its current mode/state. (General Error Code value = $0C_{hex}$)
Receive_Data	Not applicable	Discard the message	Discard the message	If a complete, valid[2] message has been received, reset the Inactivity/Watchdog Timer[3] and deliver the I/O Message to the Application. A Connection object shall exhibit the externally visible behavior associated with the current state of its attributes (see Access Rules). If this is a fragmented portion of an I/O Message, process as specified by subnet.	Discard the message

Event	I/O Connection object state				
	Non-Existent	**Configuring**	**Waiting for Connection ID**	**Established**	**Timed Out**
Send_Message	Not applicable	Return internal error - do not send the message	Return internal error - do not send the message	Transmit the complete I/O Message or fragment as required by the subnet. If this is a Client Connection and the expected_packet_rate attribute is non-zero, restart the Transmission Trigger Timer.	Return internal error - do not send the message
Inactivity/ Watchdog Timer expires	Not applicable	Not applicable	Not applicable	Examine the watchdog_timeout_action attribute of the Connection and perform the indicated action. If the watchdog_timeout_action attribute indicates that the Connection is to remain in the Established state (*Auto Reset*), then immediately re-start the Inactivity/Watchdog Timer.	Not applicable

[a] If the configuration indicates that a Message ID needs to be allocated and an available Message ID does not exist in the specified Message Group, then an Error Response whose General Error Code indicates *Resource Unavailable* (02_{hex}) is returned. If a Connection object attribute value passed the range check when it was initially configured but the attribute value conflicts with another piece of information in the node when the Apply request is processed, then an Error Response is returned whose General Error Code is set to *Invalid Attribute Value* (09_{hex}) and whose Additional Code is set to the Attribute ID of the *offending* Connection object Attribute ID.

[b] The Connection object verifies that the length of the received I/O Message is less than or equal to the consumed_connection_size attribute prior to processing the message. If the length of the received message is less than or equal to the consumed_connection_size attribute, then the I/O Connection object resets the Inactivity/Watchdog Timer, exhibits the externally visible behavior indicated by its attribute settings, and delivers the message to the Application. If the length of the received message is greater than the consumed_connection_size attribute, then the I/O Connection object immediately discards the message and discontinues any subsequent processing. This is the only message content validation performed by an I/O Connection object. Subsequent validation shall be performed by the Application.

[c] If a fragmented message is being received, then the Inactivity/Watchdog Timer is not reset until the entire message has been validly received.

NOTE Attribute Access Rules are specified in IEC 61158-5-2, 6.2.3.2.1.4

Important: The Receive_Data event is only delivered to an I/O Connection when a message whose Connection Identifier Field matches the consumed_connection_id attribute is received. If a message is received whose Connection Identifier Field does not match any Established Connection object's consumed_connection_id attribute, then the message is discarded. Connection Identifiers are subnet-type specific.

If an implementation detects that it does not support an Explicit Messaging Service indicated in Table 193, then an Error Response specifying *Service Not Supported* (General Error Code 08) is returned.

7.2.2 Bridged Connection instance behavior

Figure 26 provides a general overview of the behavior associated with a **Bridged** Connection object (**instance_type** attribute = Bridged). Bridged connections are used to make connections *offlink*. Both I/O and Explicit Messaging can be accomplished using this

connection type. The Connection Manager object definition provides more details these types of connections.

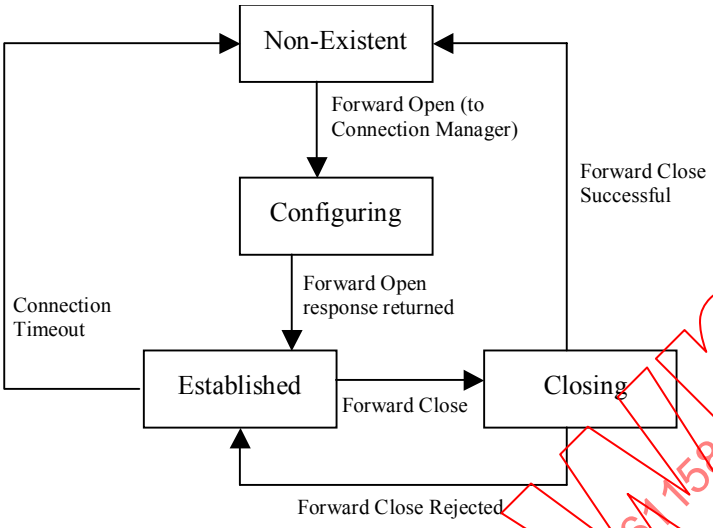


Figure 26 –Bridged Connection object state transition diagram

Table 194 provides a detailed State Event Matrix for a Bridged Connection object.

Table 194 – Bridged Connection state event matrix

Event	Bridged Connection object state			
	Non-Existent	**Configuring**	**Established**	**Closing**
Connection Manager receives a Forward Open Request	Connection Class instantiates a Connection object. Set **instance_type** to **Bridged**. Set attributes to value delivered by Forward Open service or default for Bridged connections. Transition to **Configuring.**	Not applicable	Not applicable	Not applicable
Connection Manager receives notification that connection establishment is complete to target	Not applicable	Transition to **Established**	Not applicable	Not applicable
Connection Manager receives a Forward Close Request	Not applicable	Ignore event	Transition to **Closing**	Ignore event
Connection Manager receives a Forward Close Response	Not applicable	Release all resources and transition to **Non-Existent**	Release all resources and transition to **Non-Existent**	Release all resources and transition to **Non-Existent**
Delete	Error: Object does not exist (General Error Code 16_{hex})	Release all resources and transition to **Non-Existent**	Release all resources and transition to **Non-Existent**	Release all resources and transition to **Non-Existent**
Get_Attribute_Single	Error: Object does not exist (General Error Code 16_{hex})	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.
Set_Attribute_Single	Error: Object does not exist (General Error Code 16_{hex})			
Reset	Error: Object does not exist (General Error Code 16_{hex})	Error: Service Not Supported (General Error Code 08_{hex})	Error: Service Not Supported (General Error Code 08_{hex})	Error: Service Not Supported (General Error Code 08_{hex})
Apply_Attributes	Error: Object does not exist (General Error Code 16_{hex})	Error: Service Not Supported (General Error Code 08_{hex})	Error: Service Not Supported (General Error Code 08_{hex})	Error: Service Not Supported (General Error Code 08_{hex})
Receive_Data	Not applicable	Ignore event	Invoke send service of Connection object on destination port, passing the received data.	Ignore event
Send_Message	Not applicable	Ignore event	Send data on subnet.	Ignore event
Inactivity/Watchdog Timer expires	Not applicable	Not applicable	Release all resources and transition to **Non-Existent**	Release all resources and transition to **Non-Existent**
NOTE Attribute Access Rules are specified in IEC 61158-5-2, 6.2.3.2.1.4				

7.2.3 Explicit Messaging Connection instance behavior

Figure 27 provides a general overview of the behavior associated with an **Explicit Messaging Connection object** (**instance_type** attribute = Explicit Messaging).

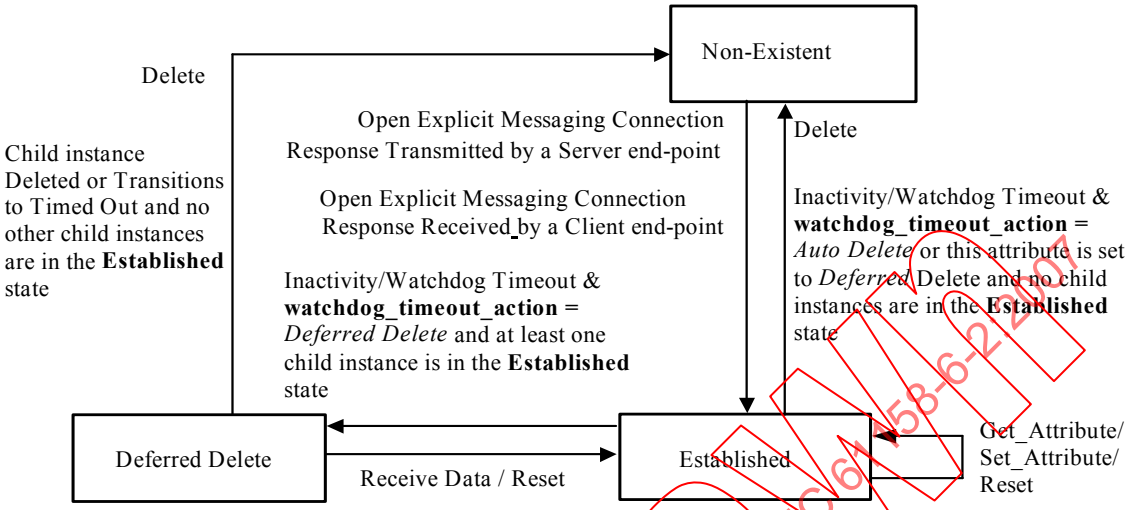


Figure 27 – Explicit Messaging Connection object state transition diagram

Table 195 provides a detailed State Event Matrix for an Explicit Messaging Connection object. Implementations should be based on the information in Table 195.

Table 195 – Explicit Messaging Connection state event matrix

Event	Explicit Messaging Connection object state		
	Non-Existent	**Established**	**Deferred Delete**
UCMM receives an Open Explicit Messaging Connection Request/Response and invokes the Create service of the Connection Class	If possible, Class instantiates Connection object. Set **instance_type** attribute to *Explicit Messaging*[a]. Other attributes are automatically configured using the information in the Open Explicit Messaging Connection Request/Response. Transition to **Established.** If request received, Transmit Open Explicit Messaging Connection Response.	Not applicable	Not applicable
UCMM receives a Close Request and invokes the Delete service of the Connection Class	Error: Object does not exist (General Error Code 16_{hex})	Release all associated resources. Transition to **Non-existent.**	Release all associated resources. Transition to **Non-existent.**
Connection Class receives a Delete Request	Error: Object does not exist (General Error Code 16_{hex})	Release all associated resources. Transition to **Non-existent.**	Release all associated resources. Transition to **Non-existent.**
Set_Attribute_Single	Error: Object does not exist (General Error Code 16_{hex})	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.
Get_Attribute_Single	Error: Object does not exist (General Error Code 16_{hex})	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.	Validate/service the request based on internal logic and per the Access Rules. Return appropriate response.
Reset	Error: Object does not exist (General Error Code 16_{hex})	Cancel the current Inactivity/Watchdog Timer. Using the value in the **expected_packet_rate** attribute, re-start the Inactivity/Watchdog Timer.	Using the value in the **expected_packet_rate** attribute, re-start the Inactivity/Watchdog Timer **and transition back to the Established state.**
Apply_Attributes	Error: Object does not exist (General Error Code 16_{hex})	Error: The object cannot perform the requested service in its current mode/state. (General Error Code value = 0Chex)	Error: The object cannot perform the requested service in its current mode/state. (General Error Code value = 0Chex)
Receive_Data	Not applicable	If a valid message or message fragment has been received, then reset the Inactivity/Watchdog Timer[b]. Either process/store the fragment or handle the Explicit Message.	If a valid message or message fragment has been received, then restart the Inactivity/Watchdog Timer[2] and transition back to the **Established** state. Either process/store the fragment or handle the Explicit Message.

Event	Explicit Messaging Connection object state		
	Non-Existent	**Established**	**Deferred Delete**
Send_Message	Not applicable	Examine the length of the message to transmit and, if necessary, perform a fragmented series of transmissions. Otherwise, transmit the complete Explicit Message.	Examine the length of the message to transmit and, if necessary, perform a fragmented series of transmissions. Otherwise, transmit the complete Explicit Message.
Inactivity/Watchdog Timer expires	Not applicable	If the **watchdog_timeout_action** attribute is set to *Auto Delete* or is set to *Deferred Delete* and no child connection instances are in the **Established** state release all associated resources and Transition to **Non-existent**. If the **watchdog_timeout_action** attribute is set to *Deferred Delete* and at least one child connection instance is in the **Established** state transition to **Deferred Delete**	Not applicable.
Child connection instance Deleted or Transitions to Timed Out	Not applicable	Ignore event	If no other child connection instances are in the **Established** state release all associated resources and transition to **Non-existent.**

[a] If the configuration indicates that a connection resource needs to be allocated and an available resource does not exist, then an Error Response whose General Error Code indicates Resource Unavailable (02hex) is returned.

[b] On CP 2/3, the MAC ID field within the Message Header of all Explicit Messages and/or Message Fragments is examined. If the Destination MAC ID is specified in the Connection ID (CAN Identifier Field), then the Source MAC ID of the other end-point shall be specified in the Message Header. If the Source MAC ID is specified in the Connection ID, then the receiving module's MAC ID shall be specified in the Message Header. If either of these checks fail, then the Inactivity/Watchdog Timer IS NOT reset and the message/message fragment is discarded.

NOTE Attribute Access Rules are specified in IEC 61158-5-2, 6.2.3.2.1.4

Important: The Receive_Data event is only delivered to an Explicit Messaging Connection when a message who's Connection ID matches the consumed_connection_id attribute is received. If a message is received whose connection id does not match any Established Connection object's consumed_connection_id attribute, then the message is discarded.

If an implementation detects that it does not support an Explicit Messaging Service indicated in Table 195, then an Error Response specifying *Service Not Supported* (General Status Code 08) is returned.

8 FAL service protocol machine (FSPM)

8.1 General

This fieldbus FAL Service Protocol State Machine maps FAL User services onto services of communication objects internal to FAL.

8.2 Primitive definitions

The Objects within FSPM shall provide the following services :

Get_Attribute_All	Reads values of all attributes of the specified object class or instance
Set_Attribute_All	Writes specified values to all attributes of the specified object class or instance
Get_Attribute_List	Reads values of the specified list of attributes of the specified object class or instance
Set_Attribute_List	Writes specified values to the specified list of attributes of the specified object class or instance
Get_Attribute_Single	Reads value of the specified attribute of the specified object class or instance
Set_Attribute_Single	Writes specified value to the specified attribute of the specified object class or instance
Reset	Resets the specified object class or instance
Create	Creates an instance of the specified object class
Delete	Deletes an instance of the specified object class
Start	Starts execution of the specified object
Stop	Stops execution of the specified object
Find_Next_Object_Instance	Finds the identifier of the next unused instance of the specified object class
NOP	Triggers corresponding NOP response from the specified object
Apply_Attributes	Causes pending attribute values to become active in the specified object
Save	Saves attributes contents of the specified object
Restore	Restores attributes contents of the specified object
Group_Sync	Verifies that each member of a group is synchronized to System Time

Refer to IEC 61158-5-2 for detailed definition of these services.

Primitives of Common Services exchanged between UCMM and FAL User are shown in Table 197 and Table 198.

All services have the following common parameters, as shown in Table 196.

Table 196 – Primitives issued by FAL user to FSPM

Common parameters	Req	Ind	Rsp	Cnf
Argument	M			
AREP	M			
Local	S			
UCMM Record identifier	S			
Transport identifier	S			
Receiver/Server Local ID		M		
Path	M	C		
Routing Path	M			
Add. Path	U	U(=)		
Port ID		M		
Result (+)			S	S(=)
AREP				M=
Receiver/Server Local ID			M=	
Service status			M	M(=)
Result (-)			S	S(=)
AREP				M=
Receiver/Server Local ID			M=	
Service status			M	M(=)

Only those argument parameters additional to the common ones are shown in Table 197 and Table 198.

Additional parameters of indication service primitives are the same as those of the corresponding request primitive.

Additional parameters of confirmation service primitives are the same as those of the corresponding response primitive.

Table 197 – Primitives issued by FAL user to FSPM

Primitive name	Source	Additional parameters	Functions
Get_attribute_all.req	FAL User	None	Conveys a request from FAL User to a target object to supply values of all attributes of the specified object class or instance
Set_attribute_all.req	FAL User	Attribute Data	Conveys a request from FAL User to a target object to write specified values of all attributes of the specified object class or instance
Get_attribute_list.req	FAL User	Attribute Count Attribute List	Conveys a request from FAL User to a target object to supply values of specified list of attributes of the specified object class or instance
Set_attribute_list.req	FAL User	Attribute Count Attribute Data	Conveys a request from FAL User to a target object to write specified values of specified list of attributes of the specified object class or instance
Get_attribute_single.req	FAL User	None	Conveys a request from FAL User to a target object to supply values of the specified attribute of the specified object class or instance
Set_attribute_single.req	FAL User	Attribute Data	Conveys a request from FAL User to a target object to write the specified values into specified attribute of the specified object class or instance
Reset.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to reset specified target object class or instance
Create.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to create an instance of the specified object class
Delete.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to delete an instance of the specified object class
Start.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to start an instance of the specified object class
Stop.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to stop an instance of the specified object class
Find_next_object_instance.req	FAL User	Maximum Returned Values	Conveys a request from FAL User to a target device to find the identifier of the next unused instance of the specified object class
NOP.req	FAL User	None	Conveys a request from FAL User to a target object to send back a corresponding NOP response
Apply_Attributes.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to activate pending attribute values
Save.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to save its attributes contents
Restore.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to restore its attributes contents
Group_Sync.req	FAL User	Object Specific Data (optional)	Conveys a request from FAL User to a target object to verify that each member of a group is synchronized to System Time
Get_attribute_all.rsp	FAL User	(+) Attribute Data (-) Specific Status Code	Returns a response from FAL User to a target object to supply values of all attributes of the specified object class or instance
Set_attribute_all.rsp	FAL User	(+) None (-) Specific Status Code	Returns a response from FAL User to a target object to write specified values of all attributes of the specified object class or instance
Get_attribute_list.rsp	FAL User	(+) Attribute Count Attribute Data (-) Specific Status Code	Returns a response from FAL User to a target object to supply values of specified list of attributes of the specified object class or instance
Set_attribute_list.rsp	FAL User	(+) Attribute_count Attribute Status List (-) Specific Status Code	Returns a response from FAL User to a target object to write specified values of specified list of attributes of the specified object class or instance
Get_attribute_single.rsp	FAL User	(+) Attribute Data (-) Specific Status Code	Returns a response from FAL User to a target object to supply values of the specified attribute of the specified object class or instance

Primitive name	Source	Additional parameters	Functions
Set_attribute_single.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to write the specified values into specified attribute of the specified object class or instance
Reset.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that the instance of the specified object class has been reset
Create.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that the instance of the specified object class has been created
Delete.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that the instance of the specified object class has been deleted
Start.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that the instance of the specified object class has started
Stop.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that the instance of the specified object class has stopped
Find_next_object_ instance.rsp	FAL User	(+) Number Of List Members Instance ID List (-) Specific Status Code	Returns a response from FAL User to a target device to find the identifier of the next unused instance of the specified object class
NOP.rsp	FAL User	None	Returns a response from FAL User to a target object to confirm that the NOP has been received
Apply_Attributes.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that pending attribute values have been activated
Save.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that its attributes contents have been saved
Restore.rsp	FAL User	(+) Object Specific Data (optional) (-) Specific Status Code	Returns a response from FAL User to a target object to confirm that its attributes contents have been restored
Group_Sync.rsp	FAL User	(+) IsSynchronized Object Specific Data (optional) (-) Specific Status Code	Conveys a request from FAL User to a target object to verify that each member of a group is synchronized to System Time

Table 198 – Primitives issued by FSPM to FAL user

Primitive name	Source	Additional parameters	Functions
Get_attribute_all.ind	FSPM	same as in .req primitive	Indicates reception of Get_attribute_all.req
Set_attribute_all.ind	FSPM	same as in .req primitive	Indicates reception of Set_attribute_all.req
Get_attribute_list.ind	FSPM	same as in .req primitive	Indicates reception of Get_attribute_list.req
Set_attribute_list.ind	FSPM	same as in .req primitive	Indicates reception of Set_attribute_list.req
Get_attribute_single.ind	FSPM	same as in .req primitive	Indicates reception of Get_attribute_single.req
Set_attribute_single.ind	FSPM	same as in .req primitive	Indicates reception of Set_attribute_single.req
Reset.ind	FSPM	same as in .req primitive	Indicates reception of Reset.req
Create.ind	FSPM	same as in .req primitive	Indicates reception of Create.req
Delete.ind	FSPM	same as in .req primitive	Indicates reception of Delete.req
Start.ind	FSPM	same as in .req primitive	Indicates reception of Start.req
Stop.ind	FSPM	same as in .req primitive	Indicates reception of Stop.req
Find_next_object_ instance.ind	FSPM	same as in .req primitive	Indicates reception of Find_next_object_instance.req
NOP.ind	FSPM	same as in .req primitive	Indicates reception of NOP.req
Apply_Attributes.ind	FSPM	same as in .req primitive	Indicates reception of Apply_Attributes.req
Save.ind	FSPM	same as in .req primitive	Indicates reception of Save.req
Restore.ind	FSPM	same as in .req primitive	Indicates reception of Restore.req
Group_Sync.ind	FSPM	same as in .req primitive	Indicates reception of Group_Sync.req
Get_attribute_all.cnf	FSPM	same as in .rsp primitive	Indicates reception of Get_attribute_all.rsp
Set_attribute_all.cnf	FSPM	same as in .rsp primitive	Indicates reception of Set_attribute_all.rsp
Get_attribute_list.cnf	FSPM	same as in .rsp primitive	Indicates reception of Get_attribute_list.rsp
Set_attribute_list.cnf	FSPM	same as in .rsp primitive	Indicates reception of Set_attribute_list.rsp
Get_attribute_signle.cnf	FSPM	same as in .rsp primitive	Indicates reception of Get_attribute_single.rsp
Set_attribute_single.cnf	FSPM	same as in .rsp primitive	Indicates reception of Set_attribute_single.rsp
Reset.cnf	FSPM	same as in .rsp primitive	Indicates reception of Reset.rsp
Create.cnf	FSPM	same as in .rsp primitive	Indicates reception of Create.rsp
Delete.cnf	FSPM	same as in .rsp primitive	Indicates reception of Delete.rsp
Start.cnf	FSPM	same as in .rsp primitive	Indicates reception of Start.rsp
Stop.cnf	FSPM	same as in .rsp primitive	Indicates reception of Stop.rsp
Find_next_object_ instance.cnf	FSPM	same as in .rsp primitive	Indicates reception of Find_next_object_instance.rsp
NOP.cnf	FSPM	same as in .rsp primitive	Indicates reception of NOP.rsp
Apply_Attributes.cnf	FSPM	same as in .rsp primitive	Indicates reception of Apply_Attributes.rsp
Save.cnf	FSPM	same as in .rsp primitive	Indicates reception of Save.rsp
Restore.cnf	FSPM	same as in .rsp primitive	Indicates reception of Restore.rsp
Group_Sync.cnf	FSPM	same as in .rsp primitive	Indicates reception of Group_Sync.rsp

8.3 Parameters of primitives

The parameters used with the primitives exchanged between the FSPM and the ARPM are described in Table 199.

Table 199 – Parameters used with primitives exchanged between FAL user and FSPM

Parameter name	Description
AREP : Local identifier UCMM Record identifier Transport identifier	Identifies the entity to be used to convey the service. This parameter may use a dedicated identifier associated with a local entity, the identifier of a UCMM transaction record previously created, or the transport identifier returned by the connection establishment process and associated with the selected AR.
Receiver/Server Local ID	Generated by the Message Router ASE of the responding node. Identifies locally this invocation of the service. It is used to associate service responses with indications.
Path : Routing Path Additional path	In the request, it specifies the FAL APO or FAL APO element that is the destination of the service request. In the indication, it contains only those segments beyond the logical class segment from the service request, i.e. the optional additional information for the target APO (Add.Path).
Port ID	Indicates on which port of the device the service indication arrived.
Service status : Status Code (mandatory) Extended Status (conditional)	Provides information on the result of service execution. It is returned in all confirmed service response primitives (+ and -). Status code indicates the type of error (see IEC 61158-5-2 for details). Extended status code gives details of the status (see IEC 61158-5-2 for details).
Attribute Data	1) A stream of information containing all of the attributes. Classes/Objects which support this service shall define the format of this parameter. 2) An array of structures, predefined by the system for the given service
Attribute Count	Number of attribute numbers in the attribute list
Attribute list []	List (array) of the attribute numbers of the attributes to get from the class or object
Attribute Number	Number representing the attribute value
Attribute Status	Status information of attribute
Attribute Value	Sequence of data specific to the attribute of the object or class
Object Specific Data	Class/Instance specific parameters. Class/Instance specification shall specify the format.
Instance ID	Value assigned to identify the newly created object
Maximum Returned Values	Maximum number of Instance ID values to be returned in the response message
Number Of List Members	Number of Instance IDs specified in response message
Instance ID List	Returned Instance ID List. The Instance IDs are returned in 16 bit integer fields.
IsSynchronized	Indicates if object is synchronized to the PTP Time Master

8.4 FSPM state machines

FSPM State Machine has got only one possible state : Running.

9 Application relationship protocol machines (ARPMs)

9.1 General

This fieldbus has Application Relationship Protocol Machines for:

- connection-less application relationships,
- connection-oriented application relationships.

Connection-less relationship is of one type only.

Connection-oriented relationships fit into one of 7 transport classes:

0 Null (or Base)
1 Duplicate Detection
2 Acknowledged
3 Verified
4 Non-blocking
5 Non-blocking, Fragmenting
6 Multipoint, Fragmenting

Although similar, each of these transport classes has its own ARPM.

9.2 Connection-less ARPM (UCMM)

9.2.1 General

Functions of the connection-less ARPM are provided by the Unconnected Message Manager (UCMM) object. UCMM shall provide an unconnected request/response message services, limited to a single link that supports multiple outstanding messages. The required number of outstanding messages shall be implementation specific. The UCMM shall be present in all nodes, and shall support at least one outstanding message.

9.2.2 Primitive definitions

The UCMM object shall provide the following services:

UCMM_Create	Creates an instance of transaction record. Puts UCMM into Running state. This service is local, it does not result in a PDU being sent.
UCMM_Delete	Deletes existing transaction record. Puts UCMM into Inactive state. This service is local, it does not result in a PDU being sent. This service is local, it does not result in a PDU being sent.
UCMM_Write	Writes an item of application data into Transport PDU buffer ; this results in sending the data to a specified target.
UCMM_Abort	Aborts existing transaction.

Refer to IEC 61158-5-2 for detailed definition of these services.

Primitives exchanged between UCMM and FSPM are shown in the following Table 200 and Table 201.

Table 200 – Primitives issued by FSPM to ARPM

Primitive name	Source	Associated parameters	Functions
UCMM_Create_req	FSPM	fixed tag max retries,	Conveys a request from the FSPM to the ARPM to create a transaction record.
UCMM_Delete_req	FSPM	record	Conveys a request from the FSPM to the ARPM to delete previously established transaction record.
UCMM_Write_req	FSPM	record destination ID UCMM service response timeout transaction priority application data	Conveys a request from the FSPM to the ARPM to send an item of application data using a previously created transaction record.
UCMM_Write_rsp	FSPM	record application data	Conveys a response from the FSPM, via Message Router to the ARPM to send a response to UCMM_Send_req using a previously created transaction record.
UCMM_Abort_req	FSPM	record	Aborts the transaction corresponding to the specified transaction record ; removes the packet from the local DLL if it has not yet been transmitted.

Table 201 – Primitives issued by ARPM to FSPM

Primitive name	Source	Associated parameters	Functions
UCMM_Create_cnf	ARPM	record service_status	Conveys a confirmation from the ARPM to the FSPM that transaction record has been created.
UCMM_Write_ind	ARPM	record source ID application data	Conveys an indication from the ARPM to the Message Router that data has arrived on the previously created transaction record. The Message Router then passes the indication to the appropriate application object.
UCMM_Write_cnf	ARPM	record service status application data	Conveys a confirmation from the ARPM to the Message Router of the execution of FSPM UCMM_Send_req on a previously created transaction record.

9.2.3 Parameters of primitives

The parameters used with the primitives exchanged between the FSPM and the ARPM are described in Table 202 .

Table 202 – Parameters used with primitives exchanged between FSPM and ARPM

Parameter name	Description
record	Identifies unambiguously the instance of the temporary connection along which the FSPM has issued a UCMM_Send request primitive. A means for such identification is not specified in this standard.
application data	Conveys FAL-User data.
fixed tag	Set to value of 0x83 for UCMM and 0x88 for Management UCMM
destination ID	Unabiguous identifier (MAC ID) of the source node of UCMM_write_req.
source ID	Unabiguous identifier (MAC ID) of the destination node of UCMM_write_req.
service	The service parameter shall specify the delivery mechanism to use for this message and shall be one of : Request_With_Acknowledge = 2 (retries untill acknowledged) Request_With_No_Response = 4 (no acknowledgement) Request_With_No_ACK = 7 (retries untill response) Request_With_No_Retry = 8 (with response) NOTE These values correspond to command codes contained in the UCMM header
timeout	Duration of the transaction in microseconds. If no UCMM_Send_cnf is received before the time-out expires, the transaction is aborted and the send_status parameter shall be TIMEOUT.
retries	Conveys the maximum allowable number of retries.
create_status	Status returned with UCMM_Create_conf = - success - cannot create
send_status	Status returned with UCMM_Send_conf = - success - record_not_created - packet_too_big - no_acknowledge - aborted - timeout

9.2.4 UCMM state machines

9.2.4.1 UCMM client

9.2.4.1.1 States

The defined states and their descriptions of the UCMM Client are listed in Table 203.

Table 203 – UCMM client states

State	Description
Inactive	The record for UCMM transfer does not exist.
Running	The record for UCMM transfer has been created.
Waiting for response	Client sent an item of application data and is waiting for a response.

9.2.4.1.2 State transition diagram

Figure 28 shows the state transition diagram of UCMM client.

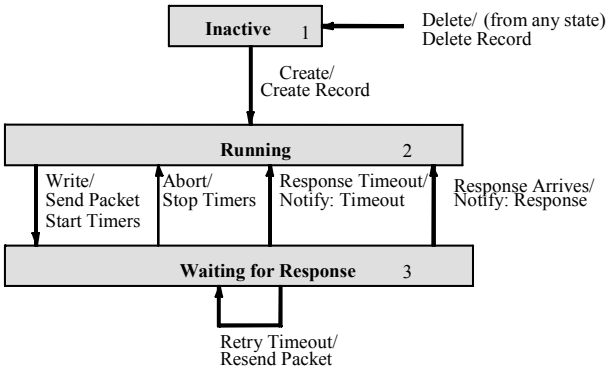


Figure 28 – State transition diagram of UCMM client

9.2.4.1.3 State event matrix

The state transitions for the UCMM Client shall be as specified in Table 204.

Table 204 – State event matrix of UCMM client

Event	State		
	Inactive	Running	Waiting for response
Create.req	1) create record 2) transition to Running		
Delete.req	no action	1) delete record 2) increment SEQUENCE 3) transition to Inactive	1) Delete record 2) increment SEQUENCE 3) transition to Inactive
Write.req	confirm status = INVALID_RECORD	1) send packet 2) start Response Timer 3) initialise Retry Count 4) start Retry Timer 5) transition to Waiting	confirm status = BUSY
Abort.req	confirm status = INVALID_RECORD	no action	1) stop timers 2) increment SEQUENCE 3) transition to Running
Retry Timeout Request packet available			1) Decrement Retry Count IF Retry Count expired 2) Notify: Timeout-No ACK 3) Free request buffer 4) Increment Sequence # 5) Stop Response Timer 6) Transition to Running ELSE. 1) Resend packet 2) Restart Retry Timer
Response Timeout			1) confirm with status = TIMEOUT-No Response 2) increment SEQUENCE 3) transition to Running
Response Arrives		no action	1) confirm with status = SUCCESS 2) Send ACK_RESP, unless response indicates no ACK_RESP desired 3) increment SEQUENCE 4) transition to Running
ACK_REQ Arrives, sequence number matches stored value		No Action	1) Free request buffer 2) stop retry timer
ACK_REQ Arrives, sequence number not equal to stored value		No Action	No Action

The sequence number shall be set to zero at initialisation. The sequence number value shall be maintained in the inactive state, to be used when the record transitions to running. The retry timeout value shall be fixed for the link at a value which guarantees that both the client and server nodes have an opportunity to transmit an unscheduled packet.

The response time-out is the response time provided when the record is created.

9.2.4.2 High-end UCMM server

9.2.4.2.1 Functions

When a packet arrives at the server an instance of a transaction shall be created if resources are available. If resources are not available the packet shall be dropped. When the instance is created, a transaction record shall be created for that instance. This record shall be active for the life of this transaction. The transaction shall end:

- when an ACK_RESP is received after a response was sent;
- when the response time timer expires; or
- when a new packet is received after a response was sent.

9.2.4.2.2 States

The defined states and their descriptions of the UCMM High-end Server are listed in Table 205.

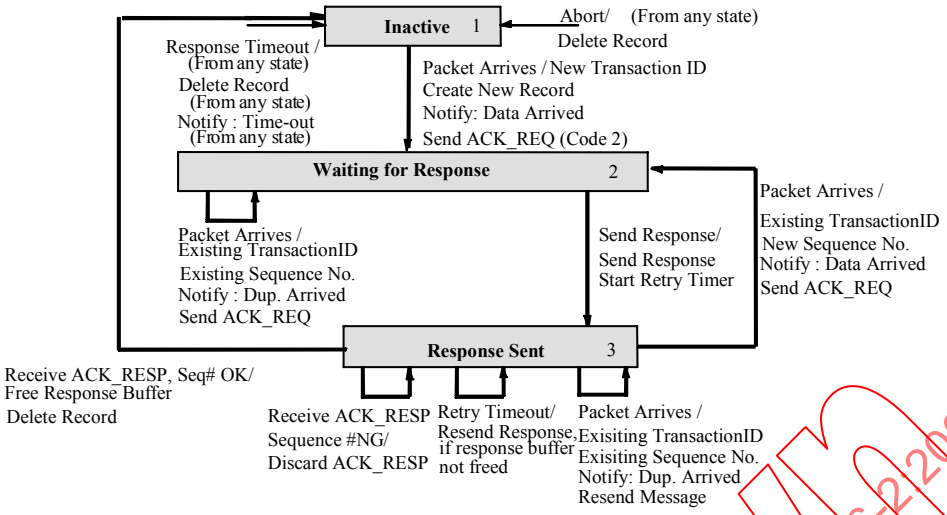
Table 205 – High-end UCMM server states

State	Description
Inactive	The record for UCMM transfer does not exist.
Waiting for Response	Data packet arrived with new Transaction ID. New record is created. The Server is waiting for response to be sent by the Server application.
Response sent	Server application sent a response.

9.2.4.2.3 State transitions

Figure 29 shows the high-end UCMM Server state transition diagram.

Figure 29 – State transition diagram of high–end UCMM server



9.2.4.2.4 State event matrix

The state transitions for the high-end UCMM server shall be as specified in Table 206.

The sequence number shall be set to zero at initialisation. The sequence number value shall be maintained in the Inactive state, to be used when the record transitions to Running. The retry time-out value shall be fixed for the local link.

The response time-out shall be the response time provided when the record is created.

Table 206 – State event matrix of high-end UCMM server

Event	State		
	Inactive	Waiting for response	Response sent
Packet arrives request (code 2) New Transaction ID and source address	1) Create new record 2) Notify: Data Arrived 3) Send ACK_REQ 4) Start Response Timer 5) Transition to Waiting for App Response	Not Applicable	Not Applicable
Packet arrives request (code 7) New Transaction ID and source address	1) Create new record 2) Notify: Data Arrived 3) Start Response Timer 4) Transition to Waiting for App Response	Not Applicable	Not Applicable
Packet arrives Existing Transaction ID and source address New Sequence #	Not Applicable	No Action	1) Update Record 2) Notify: Data Arrived 3) Send ACK_REQ 4) Transition to Waiting for App Response
Packet arrives Existing Transaction ID Existing Sequence #	Not Applicable	1) Notify: Dup. Arrived (Optional) 2) Send ACK_REQ	1) Notify: Dup. Arrived (Optional) 2) Resend Response Message
Response Timer Expires	Not Applicable	1) Notify: Timeout 2) Delete record 3) Transition to Inactive	1) Notify: Timeout 2) Delete record 3) Transition to Inactive
Abort	Error	1) Delete record 2) Transition to Inactive	1) Delete record 2) Transition to Inactive
Send Response	Error	1) Send Response 2) Start Retry Timer 3) Transition to Response Sent	Error
Retry Timeout	Not Applicable	Not Applicable	1) Decrement Retry Count IF Retry Count expired 2) Free Response buffer 3) Stop Response Timer 4) Transition to Inactive ELSE 1) Resend Response message 2) Start Retry Timer
ACK_RESP arrives, sequence number matches stored value	No action	No action	1) Delete record 2) Transition to Inactive
ACK_RESP arrives, sequence number not equal to stored value	No action	No action	No action

The response time is the response time received in the message header.

9.2.4.3 Low-end UCMM server

9.2.4.3.1 Functions

The low-end server cannot support the following features:

- perform response message retries,
- detect duplicate messages.

9.2.4.3.2 States

The defined states and their descriptions of the UCMM Low-end Server are listed in Table 207.

Table 207 – Low-end UCMM server states

State	Description
Inactive	The record for UCMM transfer does not exist.
Waiting for Response	Data packet arrived with new Transaction ID. New record is created. The Server is waiting for response to be sent by the Server application.

9.2.4.3.3 State transitions

Figure 30 shows the low-end UCMM server state transition diagram.

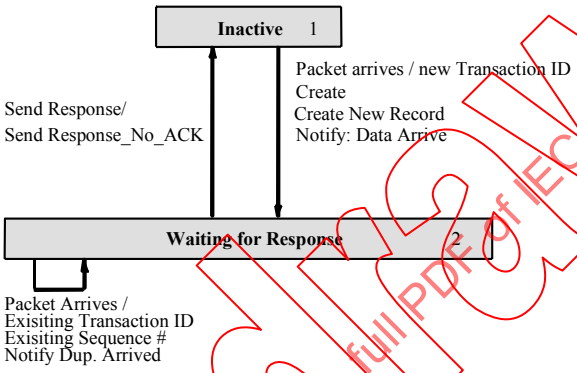


Figure 30 – State transition diagram of low–end UCMM server

9.2.4.3.4 State event matrix

The state transitions for the Low-end UCMM server shall be as specified in in Table 208.

Table 208 – State event matrix of low–end UCMM server

Event	State	
	Inactive	**Waiting for response**
Packet arrives New Transaction ID and source address	1) Create new record 2) Notify: Data Arrived 3) IF Response not immediately available, Send ACK_REQ 4) Transition to Waiting for App Response	Not Applicable
Packet arrives Existing Transaction ID and source address New Sequence #	No action	No action
Packet arrives Existing Transaction ID Existing Sequence #	No action	1) Notify: Dup. Arrived (Optional)
Send Response	Error	1) Send Response_No_ACK 2) Transition to Inactive
ACK_RESP arrives	No action	No Action

9.2.5 Examples of UCMM sequences

Figure 31 shows a sequence diagram for a UCMM with one outstanding message and Figure 32 shows a sequence diagram for a UCMM with multiple outstanding messages.

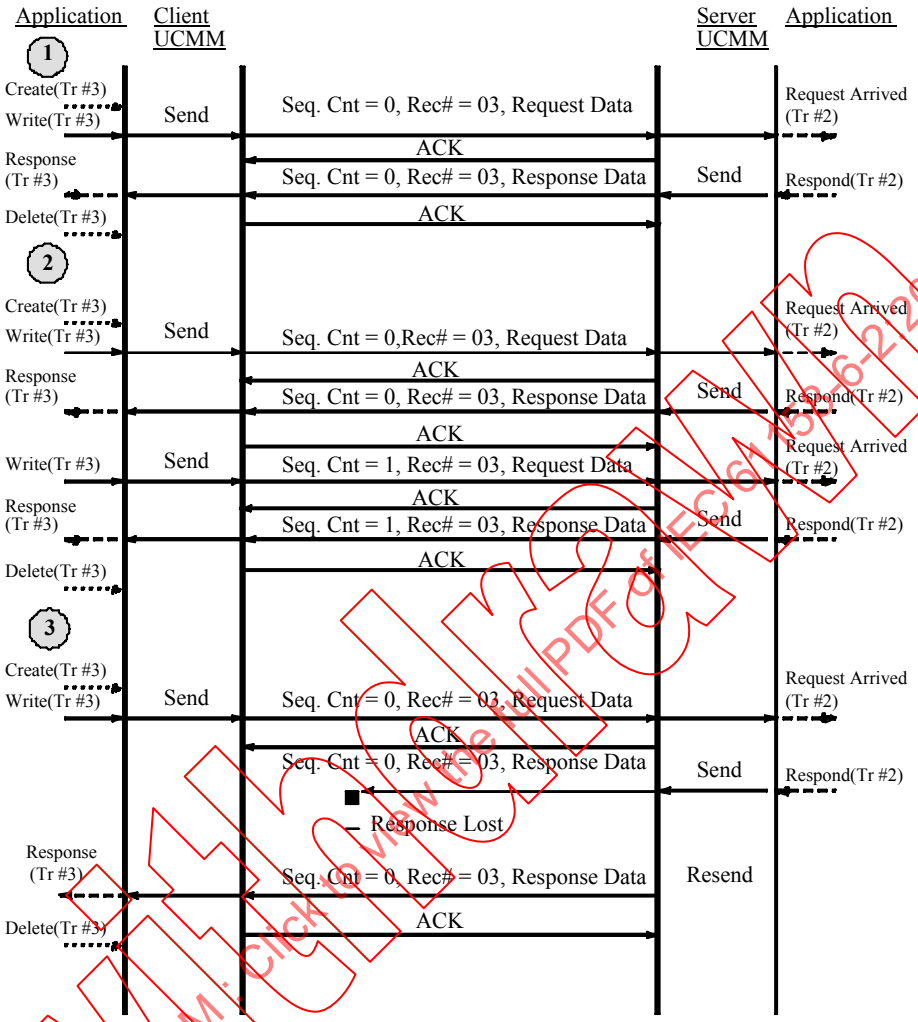


Figure 31 – Sequence diagram for a UCMM with one outstanding message

Example 1. Typical transaction: Instance is created, request is written, response is returned, and instance is deleted.

Example 2. Two transactions/same instance: In this case the instance is opened and two requests are sent.

NOTE 1 The client waits for the first response to return before issuing a second request using the same transaction number.

Example 3. Lost response: In this case the response is lost and the server issues a retry.

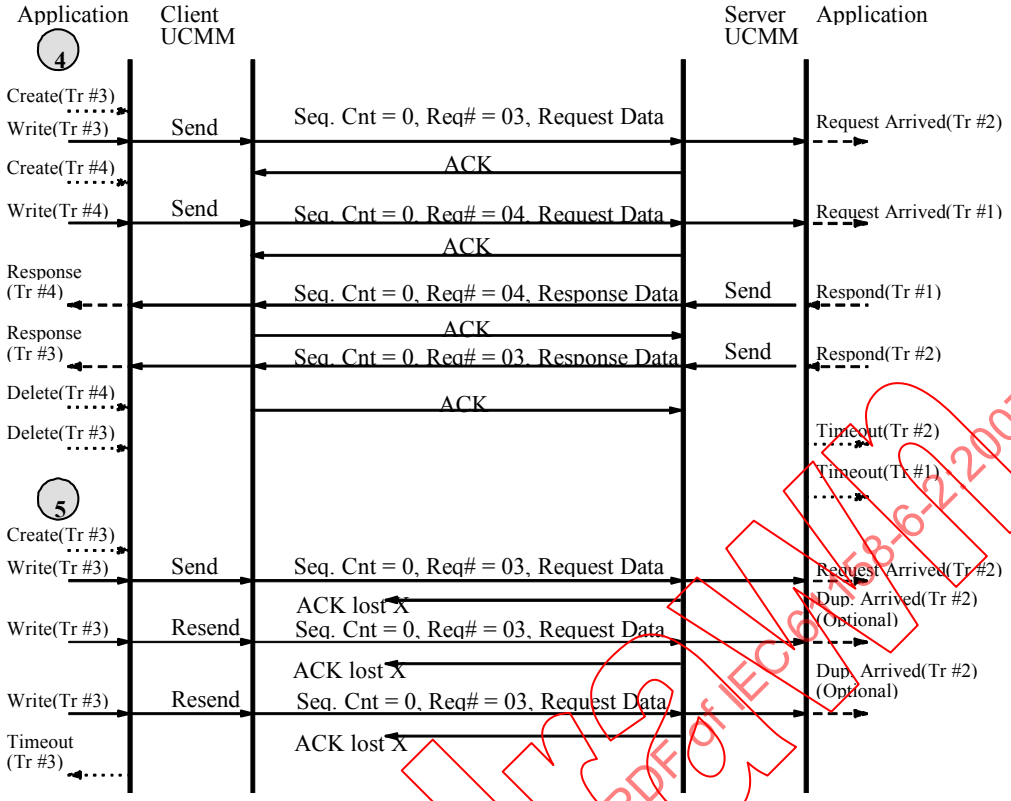


Figure 32 – Sequence diagram for a UCMM with multiple outstanding messages

Example 4. Multiple outstanding: In this case the client creates two instances and sends a second request before the first request has completed.

NOTE 2 The client uses a different transaction number for the second request.

Example 5. Timeout: In this case the server application does not respond and the client issues a timeout. The timeout causes an error status to be returned to the application. Transaction instance shall be deleted.

9.2.6 Management UCMM

Any device that has implemented the Keeper object shall also implement the Management UCMM server. With two exceptions, the Management UCMM server shall be identical to the fixed tag 0x83 UCMM server:

- transactions for the Management UCMM server shall be transmitted on fixed tag 0x88;
- the Management UCMM server shall not transmit any packets unless its node contains a Keeper object in the master state.

Any device may implement the Management UCMM client. The Management UCMM client is identical to the fixed tag 0x83 UCMM client except that transactions for the Management UCMM client shall be transmitted on fixed tag 0x88.

NOTE Messages to the Keeper object are broadcast since all devices are permitted to implement this object. To facilitate communication to the Keeper object without impacting non-Keeper devices, these messages are transmitted on a different fixed tag.

9.3 Connection-oriented ARPMs (transports)

9.3.1 Transport PDU buffer

9.3.1.1 Transport PDU format

The TPDU buffer contains a TPDU packet. The TPDU packet consists of a transport header and data. Applications shall write data to and read data from the TPDU buffers directly. Instances of transports shall write and read the transport header in the TPDU buffer while consumers shall write data to the TPDU buffer and producers shall read data from the TPDU buffer. The TPDU buffers and buffer management are not part of the communication services. This process is shown in Figure 33.

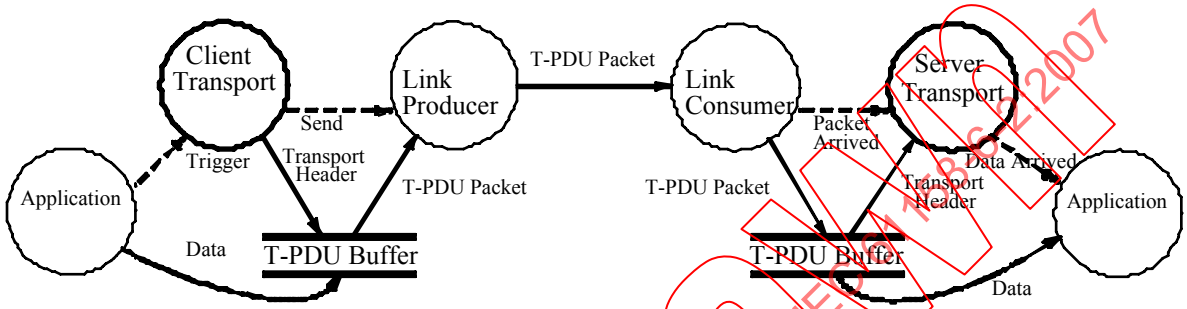


Figure 33 – TPDU buffer

9.3.1.2 Transport PDU buffer management

TPDU buffer management is implementation specific. Implementations that use single buffering, double buffering, overwrite or queuing to manage TPDU buffers are all allowed.

Implementers are responsible for ensuring buffer integrity (atomic access).

Applications are responsible for ensuring that the buffer has been initialised before a trigger is issued. Failure to initialise the buffer shall allow unknown data to be sent. This is true for producers and the client and server sides.

9.3.1.3 Notification

Transport classes can signal the application through events. The valid events are as specified in Table 209.

Table 209 – Notification

Event	Client				Server			
	Class 0	Class 1	Class 2	Class 3	Class 0	Class 1	Class 2	Class 3
Data Arrived					n	n	n	n
Duplicate Arrived						n	n	n
Acknowledgement			n					
Verification				n				

NOTE A *Duplicate Arrived* event is normally not of interest to the application; however, it might be useful in triggering a watchdog timer to indicate that the client's application is alive and triggering.

9.3.2 Transport classes

Applications interface to transport services through the supported transport classes. Table 210 lists the general–purpose transport classes that have been defined for the systems. Each transport class provides a different level of functionality. Transport class 0 provides the

minimum functionality required of a transport class, enabling data transfer between applications.

Table 210 – Transport classes

Class number	Class name
0	Null (or Base)
1	Duplicate Detection
2	Acknowledged
3	Verified
4	Non-blocking
5	Non-blocking, Fragmenting
6	Multipoint, Fragmenting

9.3.3 Common primitive definitions

Transport classes shall provide the following services:

TR_Write	Writes an item of application data into sending Transport PDU buffer, ready to be sent through the pre-established connection.
TR_Trigger	Causes an item of application data or response data previously placed in sending Transport PDU buffer, to be sent through the pre-established connection.
TR_Packet_arrived	Indicates to the Transport instance that a data packet arrived in the receiving Transport PDU buffer. This service is initiated by local action.
TR_Ack_received	Indicates to the user that a data packet arrived in the receiving Transport PDU buffer indicating an acknowledgement of arrival of previously sent data in the receiving Transport PDU buffer. This service is initiated by local action.
TR_Verify	Indicates to the user that a data packet arrived in the receiving Transport PDU buffer indicating a verification of arrival of previously sent data in the receiving user object. This service is initiated by local action.
TR_Status_Update	Indicates to the user that an updated status is present in the Transport PDU buffer.

Refer to IEC 61158-5-2 for detailed definition of these services.

Primitives exchanged between Transport instances and FSPM are shown in the following Table 211 and Table 212.

Table 211 – Primitives issued by FSPM to ARPM

Primitive name	Source	Associated parameters	Functions
TR_Write_req	FSPM	Transport identifier Application data	Conveys a request from the FSPM to the ARPM to write application data into Transport PDU buffer.
TR_Trigger.req	FSPM	Transport identifier	Conveys a request from the FSPM to the ARPM to trigger transfer of application data from Transport PDU buffer to the Link Producer.
TR_Verify.req	FSPM	Transport identifier	Conveys a request from the FSPM to the ARPM to trigger transfer of write verification data into Transport PDU buffer.

Table 212 – Primitives issued by ARPM to FSPM

Primitive name	Source	Associated parameters	Functions
TR_Packet_arrived.ind	ARPM	Transport identifier Data arrived Duplicate arrived	Conveys an indication to the FSPM that an item of data has arrived at the Link Consumer.
TR_Ack_arrived.ind	ARPM	Transport identifier	Conveys an indication to the FSPM that an acknowledgement has arrived at the Link Consumer.
TR_Status_Update.ind	ARPM	Transport identifier	Conveys an indication to the FSPM that a new status has arrived at the Link Consumer.
TR_Verify.cnf	ARPM	Transport identifier	Conveys a confirmation from the ARPM to FSPM to indicate reception of verification DLPDU.

9.3.4 Parameters of common primitives

The parameters used with the primitives exchanged between the FSPM and the ARPM are described in Table 213 .

Table 213 – Parameters used with primitives exchanged between FSPM and ARPM

Parameter name	Description
Transport identifier	Identifier of the instance of the transport invoked in the transaction.
Application data	Contains application data (service request/response parameters or user formatted data).
Data arrived Duplicate arrived	The Data_Arrived and Duplicate_Arrived parameters indicate whether the new packet contains new data, or whether it is just a duplicate from a previously received packet.

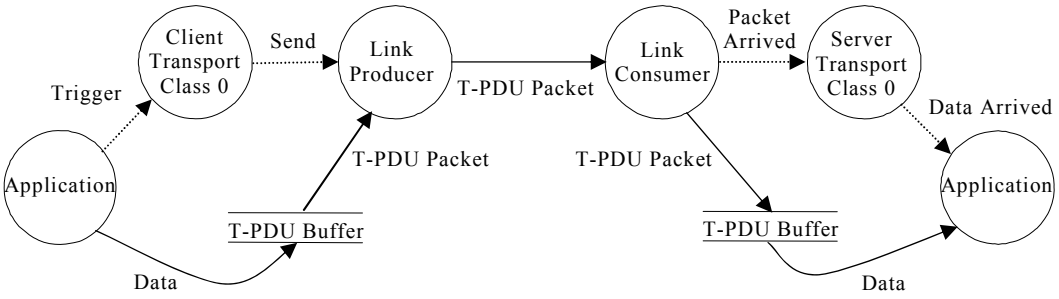
9.3.5 Transport state machines – class 0

9.3.5.1 Functions

Transport class 0 is the simplest transport class, and can use either a point–to–point or multipoint connection. Class 0 is typically used to transmit or receive inputs on a multipoint connection or outputs on a point–to–point connection.

Possible uses of transport class 0 include sampled inputs, standard outputs, cyclic block transfers, diagnostic events, and communication between controllers and operator interface devices.

Since transport class 0 does not detect duplicate data packets, the target application is responsible for detecting them. A duplicate data packet is a retransmission of the last data packet sent. Figure 34 shows the actions which take place during a data transfer using client transport class 0 and server transport class 0.



NOTE The client transport instance does not write a sequence count to the client-side T-PDU buffer, and the server transport instance does not read a sequence count from the server-side T-PDU buffer. The transport header is defined as a don't care

Figure 34 – Data flow diagram using a client transport class 0 and server transport class 0

Figure 35 shows the sequence in which actions take place when transferring data using client transport class 0 and server transport class 0.

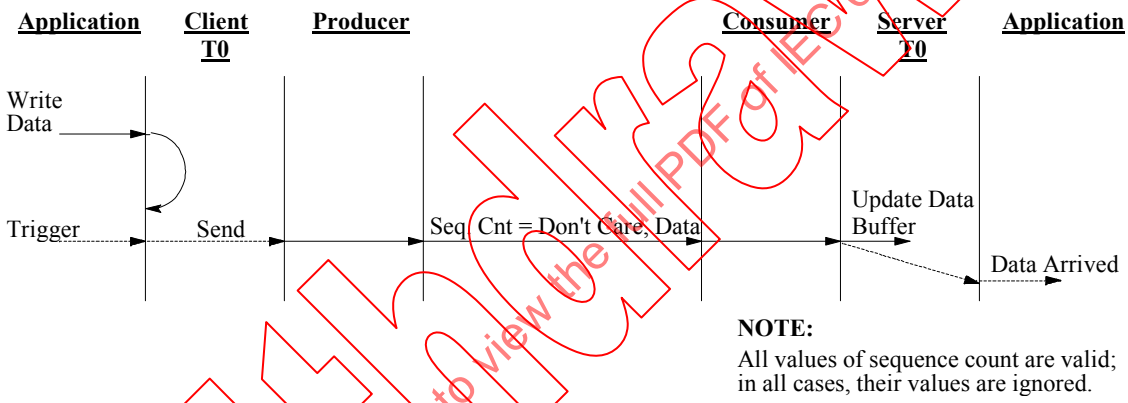


Figure 35 – Sequence diagram of data transfer using transport class 0

9.3.5.2 Transport class 0 client

9.3.5.2.1 Functions

The function of client transport class 0 is just to pass the *trigger* service from the application to the producer as a *send* service. It is useful for the client transport class 0 to be shown from a high-level design view. In actual implementations, the application may trigger the producer directly. An idle state is provided for consistency with other transport classes. In the *idle* state all events, except *delete* and *start*, are ignored.

A class 0 client transport is responsible for:

- accepting the client application's trigger that it has written a data packet to the client–side TPDU buffer,
- writing a transport header to the TPDU buffer for each packet that the client application writes to the client–side TPDU buffer (optional),
- triggering the producing node of the network connection to produce the TPDU packet on the link and send it to the consuming node of the network connection.

9.3.5.2.2 States

The defined states and their descriptions of the Class 0 Transport Client are listed in Table 214.

Table 214 – Class 0 transport client states

State	Description
Non-existent	The instance of Transport Class 0 does not exist.
Idle	The instance of Transport Class 0 has been created but not yet started.
Running	The instance of Transport Class 0 has been created and started.

9.3.5.2.3 State transitions

State transition diagram of the class 0 client transport is shown in Figure 36.

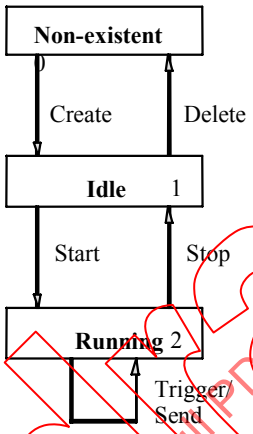


Figure 36 – Class 0 client STD

9.3.5.2.4 State event matrix

State event matrix of the class 0 client transport is shown in Table 215.

Table 215 – Class 0 client SEM

Event	State		
	Non-existent	**Idle**	**Running**
Create	Transition to **Idle**	Error	Error
Delete	Error	Transition to **Non-existent**	Error
Start	Error	Transition to **Running**	Error
Stop	Error	No Action	Transition to **Idle**
Trigger	Error	No Action	Send

9.3.5.3 Transport class 0 server

9.3.5.3.1 Function

The function of server transport class 0 is to pass the *packet arrived* event from the consumer to the producer as a *data arrived* event. It is useful for the server transport class 0 to be shown from a high-level design view. In actual implementations, the application may receive the event directly from the consumer. In the *idle* state all packet arrivals are ignored.

A class 0 server transport is responsible for:

- accepting the notification from the consuming node of the network connection that it has written a data packet to the server–side TPDU buffer,

- locating and reading the transport header of each packet that the consuming node of the network connection writes to the server–side TPDU buffer,
- notifying the server application that data has been written to the server–side TPDU buffer.

9.3.5.3.2 States

The defined states and their descriptions of the Class 0 Transport Server are listed in Table 216.

Table 216 – Class 0 transport server states

State	Description
Non-existent	The instance of Transport Class 0 does not exist.
Idle	The instance of Transport Class 0 has been created but not yet started.
Running	The instance of Transport Class 0 has been created and started.

9.3.5.3.3 State transitions

State transitions of the class 0 server transport are shown in Figure 37.

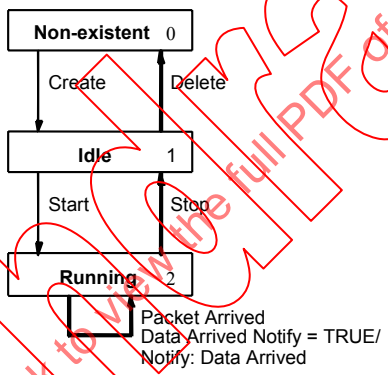


Figure 37 – Class 0 server STD

9.3.5.3.4 State event matrix

State event matrix of the class 0 server transport is shown in Table 217.

Table 217 – Class 0 server SEM

Event	State		
	Non-existent	**Idle**	**Running**
Create	Transition to **Idle**	Error	Error
Delete	Error	Transition to **Non-existent**	Error
Start	Error	Transition to **Running**	Error
Stop	Error	No action	Transition to **Idle**
Packet Arrived & Data Arrived Notify = TRUE	No action	No action	Notify: Data Arrived

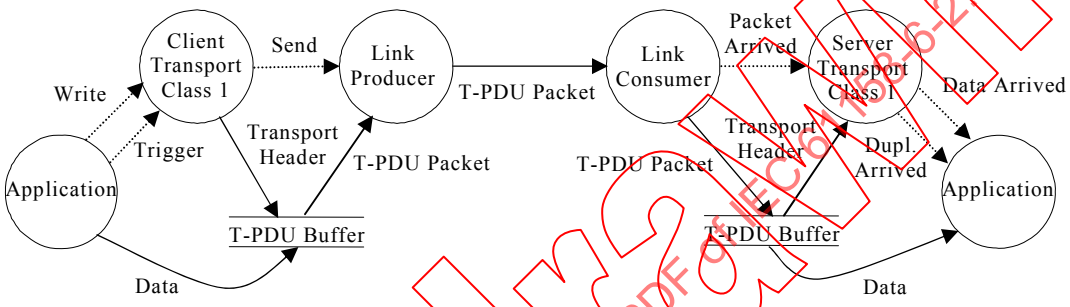
9.3.6 Transport state machines – class 1

9.3.6.1 Functions

Transport class 1, like class 0, shall allow either a point–to–point or multipoint connection. Unlike class 0, which has no transport header, the class 1 transport shall include a sequence number, which is used to detect the delivery of duplicate data packets.

NOTE 1 Class 1 is preferred to class 0 for targets because the target application does not have to perform duplicate detection, which reduces the overhead required in those end nodes that support the change of state transport trigger. Class 1 permits an efficient change of state trigger, since was designed to allow change of state data transfers.

Figure 38 shows the actions which take place during data transfer using a class 1 client transport instance and a class 1 server transport instance.



NOTE The T-PDU packet comprises the transport header from the client transport and data from the application.

Figure 38 – Data flow diagram using client transport class 1 and server transport class 1

Figure 39 shows the sequence in which actions take place when transferring data using client transport class 1 and server transport class 1.

NOTE 2 The sequence count is incremented with every write. Also, in the case where the packet is lost on a new data sample, and the packet is later triggered and sent, the data received by the client is treated as new data. This is because this is the first time the server has received this sequence count. This mechanism provides fault tolerance for those samples that change infrequently.

NOTE 3 It is an implementation choice to decide whether to update the data buffer upon receipt of duplicate data or not, based on the potential impact of the additional processing.

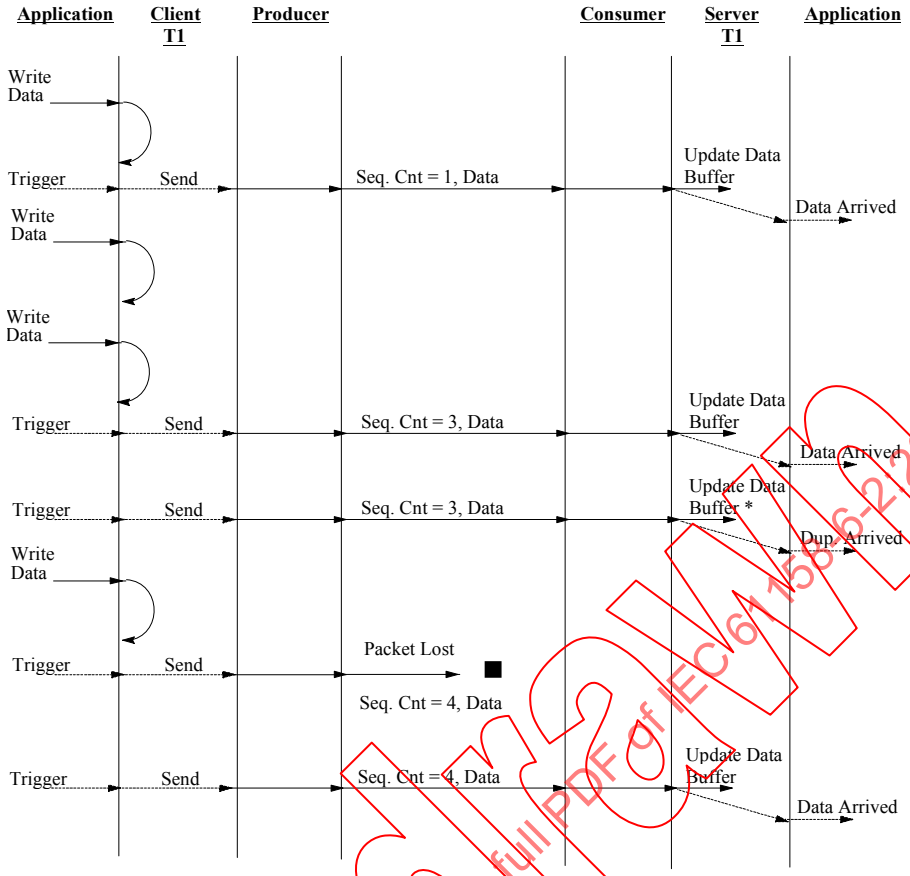


Figure 39 – Sequence diagram of data transfer using client transport class 1 and server transport class 1

Transport class 1 can be used in conjunction with a heartbeat timer to keep a connection open when there are periods when no data is transmitted. The heartbeat timer shall be initialised every time the producing node of the network connection transmits a packet, and shall be set so that its maximum value, or the maximum time between data transmissions, is less than the path time–out value for the network connection. When the heartbeat timer reaches its maximum value, it triggers the client transport instance so that the producing node produces and retransmits the packet in the client–side TPDU buffer before the connection times out. The consuming node of the network connection, recognising that the packet has the same sequence count as the last packet it received, may or may not overwrite the packet in the server–side TPDU buffer; that implementation decision is left to the product developers, since they can best assess the impact of additional processing.

Applications could use this transport class to distinguish between new samples and old samples that were sent to maintain the connection. To maintain the connection, a timer may be used to trigger the production of the current data in the TPDU buffer. To transmit new data the application would first write to the TPDU buffer and then trigger the client transport.

Applications could use this transport class to distinguish between new samples and old samples that were sent to maintain the connection. To maintain the connection, the data arrived and duplicated arrived events may be ORed together to reset a timer. If this timer were to expire, the application would know that no data was received within the designated time. The data arrived event would identify new samples of data. This may allow applications to reduce their overhead by only processing new data samples.

Possible uses of transport class 1 include sampled inputs, standard outputs, cyclic block transfers, diagnostic events, and communication between controllers and operator interface devices.

9.3.6.2 Transport class 1 client

9.3.6.2.1 Functions

Transport class 1 starts with the behaviour in class 0 and adds a sequence count. This sequence count is incremented by the client transport class when data is written to the data buffer. When the transport receives a *write* event, it shall increment the sequence count. When a *trigger* event is received the transport shall update the sequence count in the TPDU buffer and invoke the *send* service.

Applications could use this transport class to distinguish between new samples and old samples that were sent to maintain the connection. To maintain the connection, a timer may be used to trigger the production of the current data in the TPDU buffer. To transmit new data the application would first write to the TPDU buffer and then trigger the client transport.

A class 1 client transport is responsible for:

- accepting the client application's trigger that it has written a data packet to the client–side TPDU buffer,
- writing a transport header to the TPDU buffer for each packet that the client application writes to the client–side TPDU buffer,
- initialising the sequence count in the transport header (for the first packet transmitted) or incrementing the sequence count (for subsequent packets of the same transmission),
- triggering the producing node of the network connection to produce the TPDU packet on the link and send it to the consuming node of the network connection.

9.3.6.2.2 States

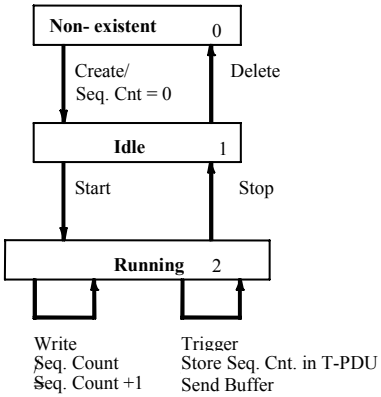
The defined states and their descriptions of the Class 1 Transport Client are listed in Table 218.

Table 218 – Class 1 transport client states

State	Description
Non-existent	The instance of Transport Class 1 does not exist.
Idle	The instance of Transport Class 1 has been created but not yet started.
Running	The instance of Transport Class 1 has been created and started.

9.3.6.2.3 State transitions

State transitions of the class 1 client transport are shown in Figure 40.



NOTE The sequence count is set to 0 by the create service. It is then incremented after every write event until it reaches a maximum value of 2^{16}-1. After that it rolls over to 0.

Figure 40 – Class 1 client STD

9.3.6.2.4 State event matrix

State transitions of the class 1 client transport are shown in Table 219.

Table 219 – Class 1 client SEM

Event	State		
	Non-existent	**Idle**	**Running**
Create	1) Transition to **Idle** 2) Set seq. count = 0	Error	Error
Delete	Error	Transition to **Non-existent**	Error
Start	Error	Transition to **Running**	Error
Stop	Error	No Action	Transition to **Idle**
Write	Error	No Action	1) Seq. Cnt = Seq. Cnt + 1
Trigger	Error	No Action	1) Store Seq. Cnt. in TPDU Buffer 2) Send

9.3.6.3 Transport class 1 server

9.3.6.3.1 Functions

Transport class 1 starts with the behaviour in class 0 and adds a sequence count. The received sequence count is compared with the previous sequence count. If they are equal, the received packet is considered to be a duplicate packet. If they are not equal, the received data is considered to be a new sample. There is no attempt to count how many sequence counts have changed between samples.

Applications could use this transport class to distinguish between new samples and old samples that were sent to maintain the connection. To maintain the connection, the data arrived and duplicated arrived events may be ORed together to reset a timer. If this timer were to expire, the application would know that no data was received within the designated time. The data arrived event would identify new samples of data. This may allow applications to reduce their overhead by only processing new data samples.

A class 1 server transport is responsible for:

- accepting the notification from the consuming node of the network connection that it has written a data packet to the server–side TPDU buffer,

- locating and reading the transport header of each packet that the consuming node of the network connection writes to the server–side TPDU buffer,
- triggering the server application that data has been written to the server–side TPDU buffer,
- comparing the sequence count of each packet with that of the previous packet,
- notifying the server application that the most recently arrived packet is a duplicate of the previous one when their sequence counts match.

9.3.6.3.2 States

The defined states and their descriptions of the Class 1 Transport Server are listed in Table 220.

Table 220 – Class 1 transport server states

State	Description
Non-existent	The instance of Transport Class 1 does not exist.
Idle	The instance of Transport Class 1 has been created but not yet started.
Ready to run	The instance of Transport Class 1 has been created and started, waiting for a first data packet to arrive.
Running	The instance of Transport Class 1 has been created and started, first data packet has arrived.

9.3.6.3.3 State transitions

State transitions of the class 1 server transport are shown in Figure 41.

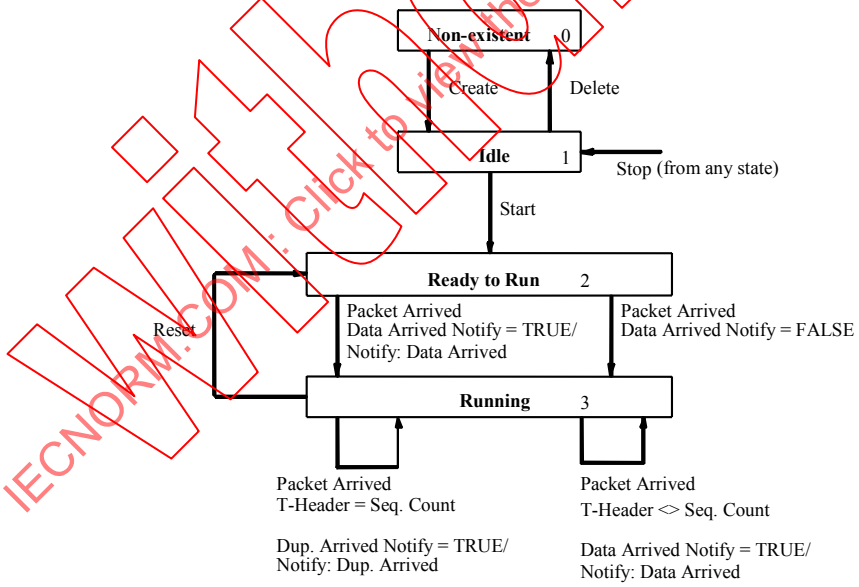


Figure 41 – Class 1 server STD

9.3.6.3.4 State event matrix

State transitions of the class 1 server transport are shown in Table 221.

Table 221 – Class 1 server SEM

Event	State/			
	Non-existent	**Idle**	**Ready to run**	**Running**
Create	Transition to **Idle**	Error	Error	Error
Delete	Error	Transition to **Non-existent**	Error	Error
Start	Error	Transition to **Ready to Run**	Error	Error
Stop	Error	No action	Transition to **Idle**	Transition to **Idle**
Reset	Error	Error	No Action	Transition to **Ready to Run**
Packet Arrived Transport Header <> Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = TRUE	No action	No action	Transition to **Running** Notify: Data Arrived	Notify: Data Arrived
Packet Arrived Transport Header = Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = TRUE	No action	No action	Transition to **Running** Notify: Data Arrived	Notify: Duplicate Arrived
Packet Arrived Transport Header <> Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = FALSE	No action	No action	Transition to **Running** Notify: Data Arrived	Notify: Data Arrived
Packet Arrived Transport Header = Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = FALSE	No action	No action	Transition to **Running** Notify: Data Arrived	No action
Packet Arrived Transport Header <> Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = TRUE	No action	No action	Transition to **Running**	No action
Packet Arrived Transport Header = Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = TRUE	No action	No action	Transition to **Running**	Notify: Duplicate Arrived
Packet Arrived Transport Header <> Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = FALSE	No action	No action	Transition to **Running**	No action
Packet Arrived Transport Header = Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = FALSE	No action	No action	Transition to **Running**	No action

9.3.7 Transport state machines – class 2

9.3.7.1 Functions

Transport class 2 starts with the behaviour in class 1 and adds a return connection that acknowledges the delivery of a packet. The producer connection from the client node can be point to point or multipoint. The connection from the server to the client is point to point.

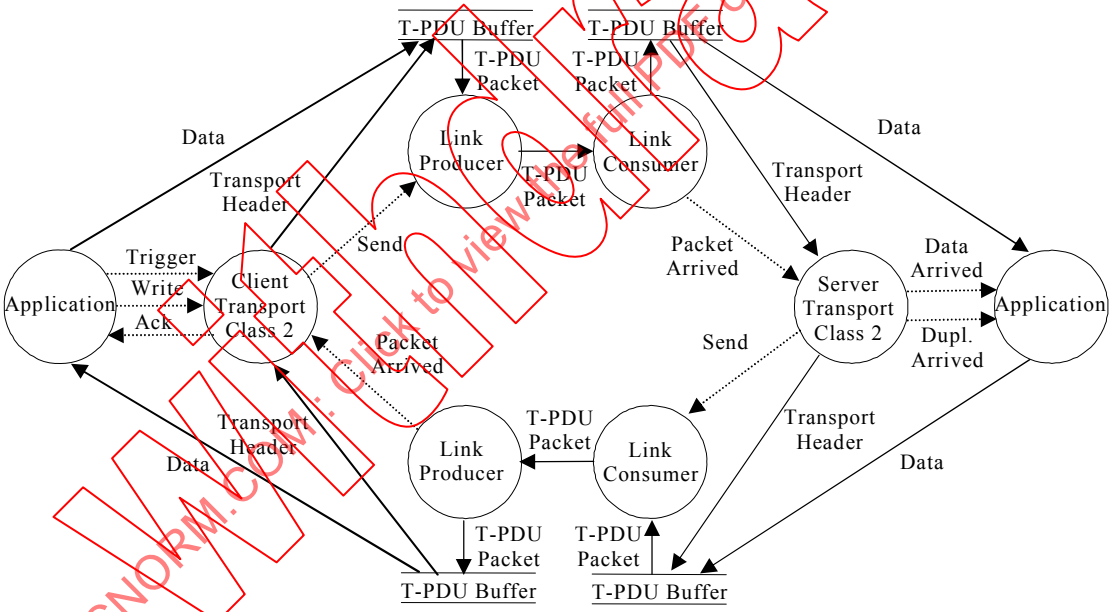
This transport class allows the client application to be notified that the server has received the transmitted packet. This acknowledgement tells the client application only that the data arrived. This does not mean that the server application has read the buffer or that a new sample can be sent without overwriting the buffer.

The main advantage of this class over class 3 (Verified) is that the acknowledgement is returned immediately after the packet arrives at the consumer, while the verify class requires the application to initiate the return verification. The delays associated with receiving the acknowledgement are small, approximately twice the time that is required for a one way transmission.

Transport class 2 uses two network connections: the originator–to–target connection, which can be either point–to–point or multipoint; and the return connection, which shall be point–to–point. A multipoint connection using transport class 2 shall have a point–to–point return connection corresponding to each client–to–server connection. *The acknowledgement notifies the client transport instance that the consuming node of the network connection has received the packet.*

If a server transport instance does not acknowledge receipt of the packet, the client transport instance triggers the client application to retransmit the most recently sent packet; if the client–to–server connection is one of the network connections of a multipoint connection, data is resent over all of the network connections of the multipoint connection. The server application can return data to the client application along with its acknowledgement.

Figure 42 shows the actions which take place during a data transfer using client transport class 2 and server transport class 2



NOTE 1 The client and server transports each use two buffers: one for transmitting, and one for receiving.

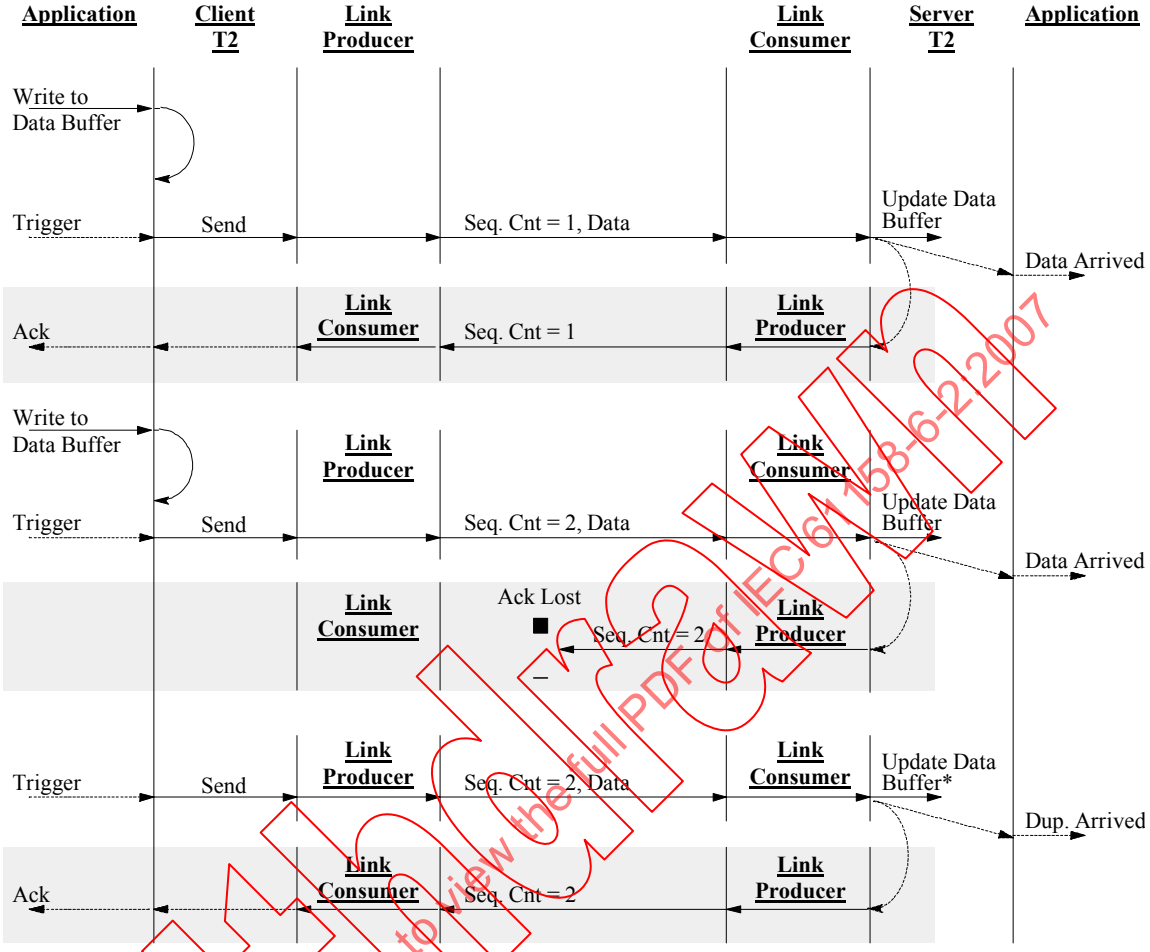
NOTE 2 The target application is not involved in sending the acknowledgement.

Figure 42 – Data flow diagram using client transport class 2 and server transport class 2

The server transport instance initiates and sends the acknowledgement to the client transport instance to notify the client application that it can write the next data packet to the client–side TPDU transmit buffer. Each acknowledgement shall be received by the client transport instance before the next packet is sent over the same client–to–server network connection(s).

Figure 43 shows the sequence of actions which take place during data transfer using client transport class 2 and server transport class 2. In the example depicted, the acknowledgement is the only data transmitted in the server–to–client direction. The unshaded areas in Figure 43

indicate client–to–server data transmission, and the shaded areas indicate server–to–client transmission.



* Implementors must decide whether to update the data buffer upon receipt of duplicate data packets, since they can best assess the impact of the additional processing.

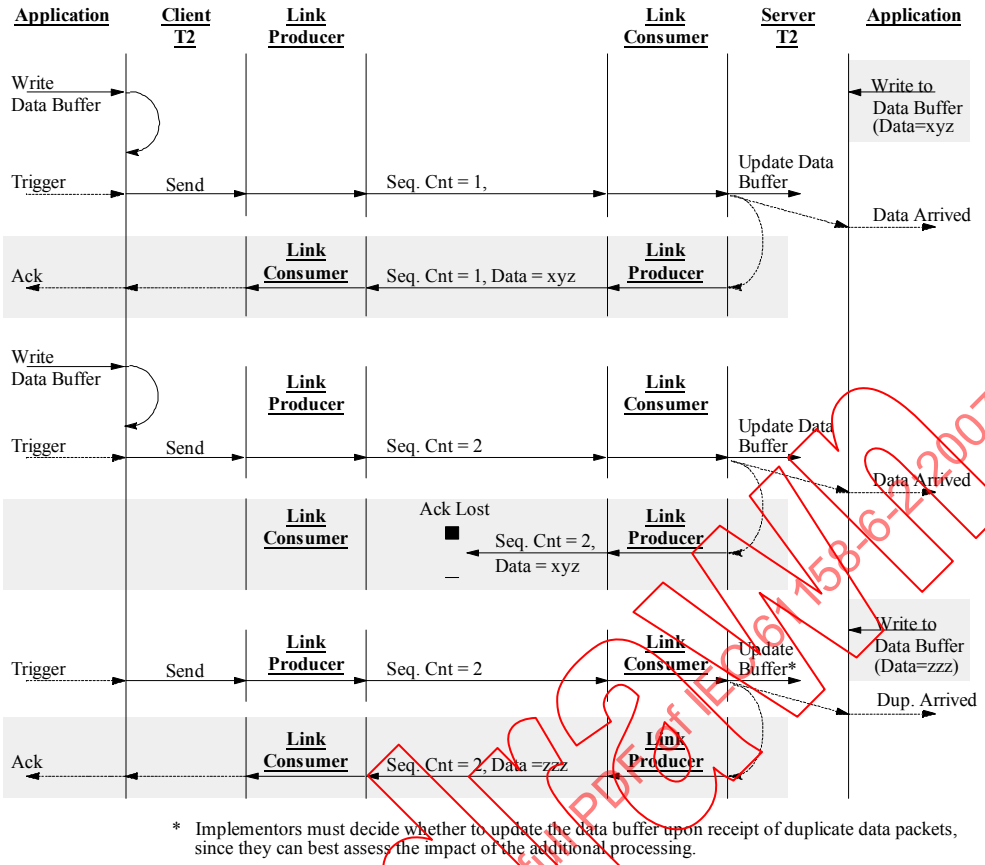
Figure 43 – Diagram of data transfer using client transport class 2 and server transport class 2 *without* returned data

At times, the server–to–client transmission comprises the acknowledgement *and* additional data.

Figure 44 shows the sequence in which actions take place during data transfer using client transport class 2 and server transport class 2 when the acknowledgement and data are returned to the client.

NOTE The returned data is written asynchronously to the receipt of data from the client.

The unshaded areas in Figure 44 indicate client–to–server data transmission, and the shaded areas indicate server–to–client data transmission.



* Implementors must decide whether to update the data buffer upon receipt of duplicate data packets, since they can best assess the impact of the additional processing.

Figure 44 – Sequence diagram of data transfer using client transport class 2 and server transport class 2 *with* returned data

Possible uses of transport class 2 include master/slave applications and communication between controllers and operator interface devices.

9.3.7.2 Transport class 2 client

9.3.7.2.1 Functions

In the client–to–server direction, a class 2 client transport is responsible for:

- accepting the client application's trigger that it has written a data packet to the client–side TPDU transmit buffer;
- writing a transport header to the client–side TPDU transmit buffer for each packet that the client application writes to it

  NOTE The transport header for each packet shall include a sequence count

- initialising the sequence count in the transport header (for the first packet transmitted) or incrementing the sequence count (for subsequent packets of the same transmission);
- triggering the producing node of the network connection to produce the TPDU packet on the link and send it to the consuming node of the network connection.

In the server–to–client direction, a class 2 client transport is responsible for:

- accepting notification from the consuming node of the return network connection that it has written data (the server transport instance's acknowledgement) to the client–side TPDU receive buffer;
- locating and reading the transport header of each data packet that the consuming node of the return network connection writes to the client–side TPDU receive buffer;

- notifying the client application that the consuming node of the return network connection has written data to the client–side TPDU receive buffer.

The application can update the data buffer or retrigger data before all of the acknowledgements have been received. If the client transport is retriggered while waiting for an acknowledgement, the previous acknowledgement received register is cleared. This means that the client transport class shall wait for all active consumers to return an acknowledgement before notifying the application.

9.3.7.2.2 States

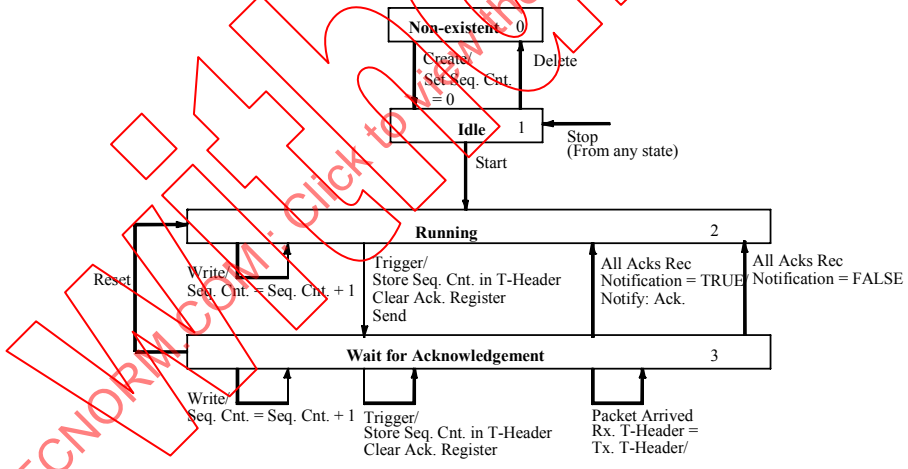
The defined states and their descriptions of the Class 2 Transport Client are listed in Table 222.

Table 222 – Class 2 transport client states

State	Description
Non-existent	The instance of Transport Class 2 does not exist.
Idle	The instance of Transport Class 2 has been created but not yet started.
Running	The instance of Transport Class 2 has been created and started.
Wait for acknowledgement	The instance of Transport Class 2 has been created and started, acknowledgement is pending.

9.3.7.2.3 State transitions

State transitions of the class 2 client transport are shown in Figure 45.



NOTE The sequence count is set to 0 by the create service. It is then incremented after every *write* event until it reaches a maximum value of 2^{16}-1. After that it rolls over to 0.

Figure 45 – Class 2 client STD

9.3.7.2.4 State event matrix

Sen
d

Update
Register

Ack.

State transitions of the class 2 client transport are shown in Table 223.

Table 223 – Class 2 client SEM

Event	State			
	Non-existent	**Idle**	**Running**	**Wait for ack**
Create	1) Seq. Cnt. = 0 2) Transition to **Idle**	Error	Error	Error
Delete	Error	Transition to **Non-existent**	Error	Error
Start	Error	Transition to **Running**	Error	Error
Stop	Error	No Action	Transition to **Idle**	Transition to **Idle**
Reset	Error	Error	No Action	Transition to **Running**
Write	Error	No Action	1) Seq. Cnt. = Seq. Cnt. + 1 2) Write TPDU	1) Seq.Cnt. = Seq.Cnt. +1 2) Write TPDU
Trigger	Error	No Action	1) Store Seq. Cnt. in TPDU 2) Clear Ack. register 3) Send 4) Transition to **Wait for Ack.**	1) Store Seq. Cnt. in TPDU 2) Clear Ack. Register 3) Send
Packet Arrived Rx. T-Header = Tx. T-Header	Error	No Action	No Action	1) Update Ack. Register
All Ack Received Ack. Notify = TRUE	Error	No Action	Error	1) Notify: Ack. 2) Transition to **Running**
All Ack Received Ack. Notify = FALSE	Error	No Action	Error	1) Transition to **Running**

NOTE The STD and SEM for client classes 2 and 3 are nearly identical. Verify has been substituted for acknowledgement.

9.3.7.3 Transport class 2 server

9.3.7.3.1 Functions

Server transport class 2 starts with the behaviour from transport class 1 and adds a return connection from the server to the client that is used to acknowledge a message. This return connection is used to identify the acknowledgement message.

After receiving a *packet arrived* event, the server transport shall read the received transport header, store it in the transmit transport header and invoke the *send* service. The application is not involved in the acknowledgement. The producer connection from the client node can be point to point or multipoint. The connection from the server to the client is point to point.

In the client–to–server direction, a class 2 server transport is responsible for:

- accepting the notification from the consuming node of the network connection that it has written a data packet to the server–side TPDU receive buffer;
- locating and reading the transport header of each packet that the consuming node of the network connection writes to the server–side TPDU receive buffer;
- comparing the sequence count of each packet in the server–side TPDU receive buffer with that of the previous packet;
- notifying the server application that the most recently arrived packet is a duplicate of the previous one when their sequence counts match;
- notifying the server application that data has been written to the server–side TPDU receive buffer. (ACK may contain server data).

In the server–to–client direction, a class 2 server transport is responsible for:

- writing the transport header (including sequence count) of the data in the server–side TPDU receive buffer to the server–side TPDU transmit buffer;
- triggering the producing node of the return network connection to produce the TPDU packet on the link and send it to the consuming node of the return network connection.

9.3.7.3.2 States

The defined states and their descriptions of the Class 2 Transport Server are listed in Table 224.

Table 224 – Class 2 transport server states

State	Description
Non-existent	The instance of Transport Class 2 does not exist.
Idle	The instance of Transport Class 2 has been created but not yet started.
Running	The instance of Transport Class 2 has been created and started.

9.3.7.3.3 State transitions

State transitions of the class 2 server transport are shown in Figure 46.

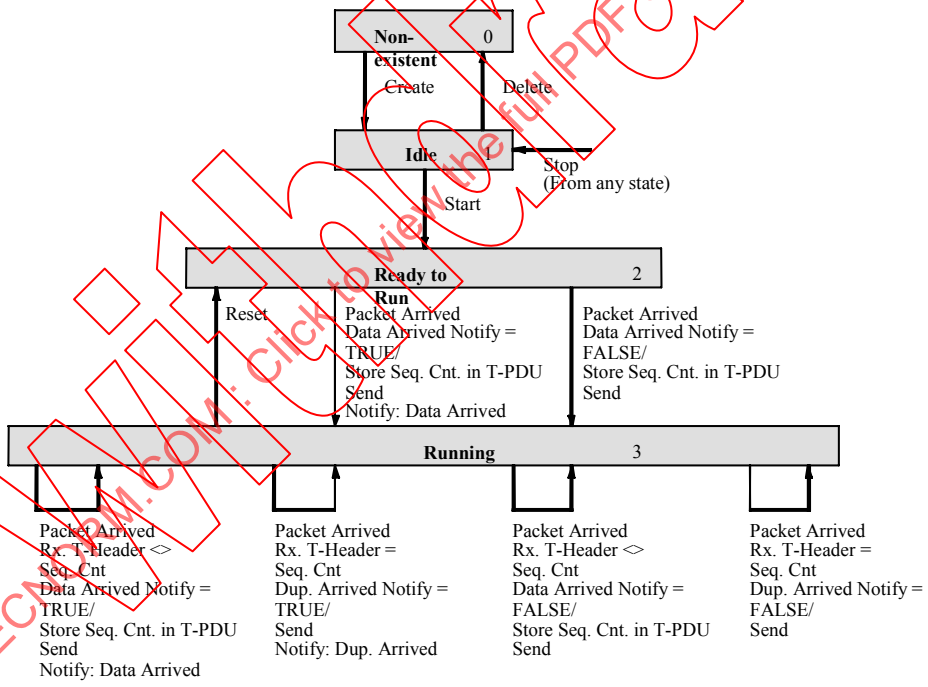


Figure 46 – Class 2 server STD

NOTE The send service is invoked for sequence counts that match and sequence counts that differ. This behaviour is required to ensure that a node that has sent an acknowledgement that was lost shall retransmit the acknowledgement on a retry.

9.3.7.3.4 State event matrix

State transitions of the class 2 server transport are shown in Table 225.

Table 225 – Class 2 server SEM

Event	State			
	Non-existent	**Idle**	**Ready to run**	**Running**
Create	Transition to **Idle**	Error	Error	Error
Delete	Error	Transition to **Non-existent**	Error	Error
Start	Error	Transition to **Ready to Run**	Error	Error
Stop	Error	No action	Transition to **Idle**	Transition to **Idle**
Reset	Error	Error	No action	Transition to **Ready to Run**
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = TRUE Duplicate Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3)Notify: Data Arrived 4) Transition to **Running**	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3)Notify: Data Arrived
Packet Arrived Rx. T-Header = Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3)Notify: Data Arrived 4) Transition to **Running**	1) Send (Implies Ack) 2) Notify: Duplicate Arrived
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = TRUE Duplicate Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3)Notify: Data Arrived 4) Transition to **Running**	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3)Notify: Data Arrived
Packet Arrived Rx. T-Header = Seq. Count Data Arrived Notify = TRUE Duplicate Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3) Notify: Data Arrived 4) Transition to **Running**	1) Send (Implies Ack)
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = FALSE Duplicate Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3) Transition to **Running**	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack)
Packet Arrived Rx. T-Header = Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3) Transition to **Running**	1) Send (Implies Ack) 2) Notify: Duplicate Arrived
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = FALSE Duplicate Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3) Transition to **Running**	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack)
Packet Arrived Rx. T-Header = Seq. Count Data Arrived Notify = FALSE Duplicate Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in Tx. TPDU 2) Send (Implies Ack) 3) Transition to **Running**	1) Send (Implies Ack)

9.3.8 Transport state machines – class 3

9.3.8.1 Functions

Transport class 3 starts with the behaviour in class 1 and adds a return connection that verifies that the application has received the packet. The sequence count that shall be returned with the verify is the same as the sequence count sent with the data. This return connection is used to identify the verification message.

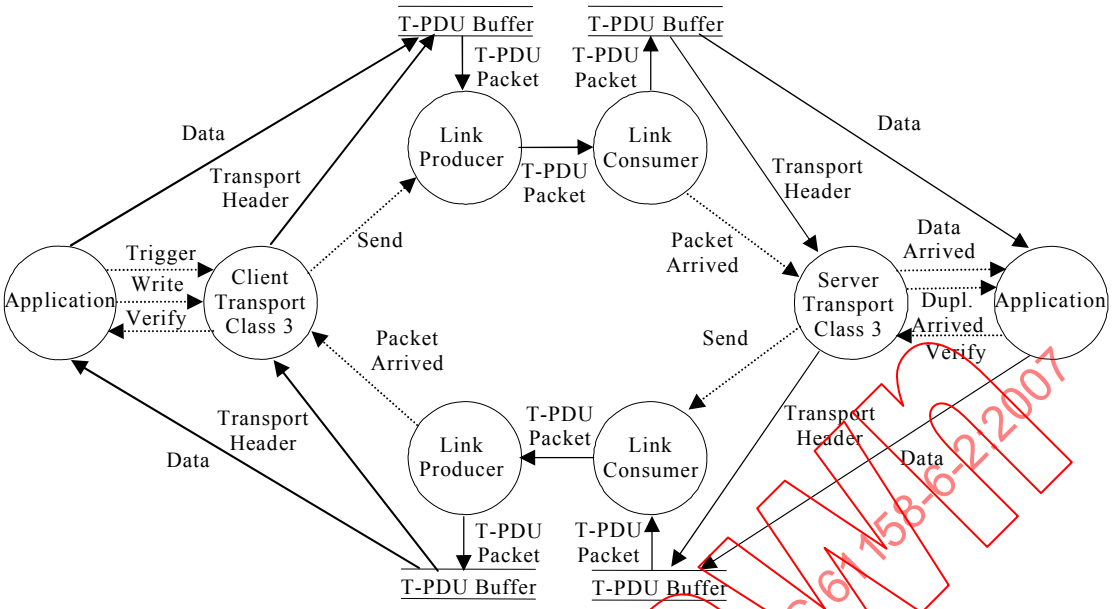
This transport class allows the application to be notified that the server application has responded to the transmitted packet. This verification tells the client application not only that the data arrived but that the server application has read the buffer and that a new sample can be sent without overwriting the buffer.

The main advantage of this class over class 2 (Acknowledgement) is that verification conveys an application response. The disadvantage is that the delay in receiving the verification can be very difficult to predict.

Transport class 3 uses two connections: the originator–to–target connection, which can be either point–to–point or multipoint; and the return connection, which shall be point–to–point. A multipoint connection using transport class 3 shall have a point–to–point return connection corresponding to each client–to–server connection. The verification notifies the client application that the server application has received and read the transmitted data. If a server application does not verify receipt of the packet, the client application retransmits the most recently sent packet; if the client–to–server connection is a multipoint connection, the data is retransmitted over all of the network connections of the multipoint connection.

The server application initiates and sends the verification *after* the consuming node of the network connection has written the packet to the TPDU buffer, *after* the transport has read the packet's transport header, and *after* the application has read the packet. The server's verification signals the client application that the server–side TPDU receive buffer is ready to accept the next data packet.

Figure 47 shows the actions which take place during a data transfer using client transport class 3 and server transport class 3.

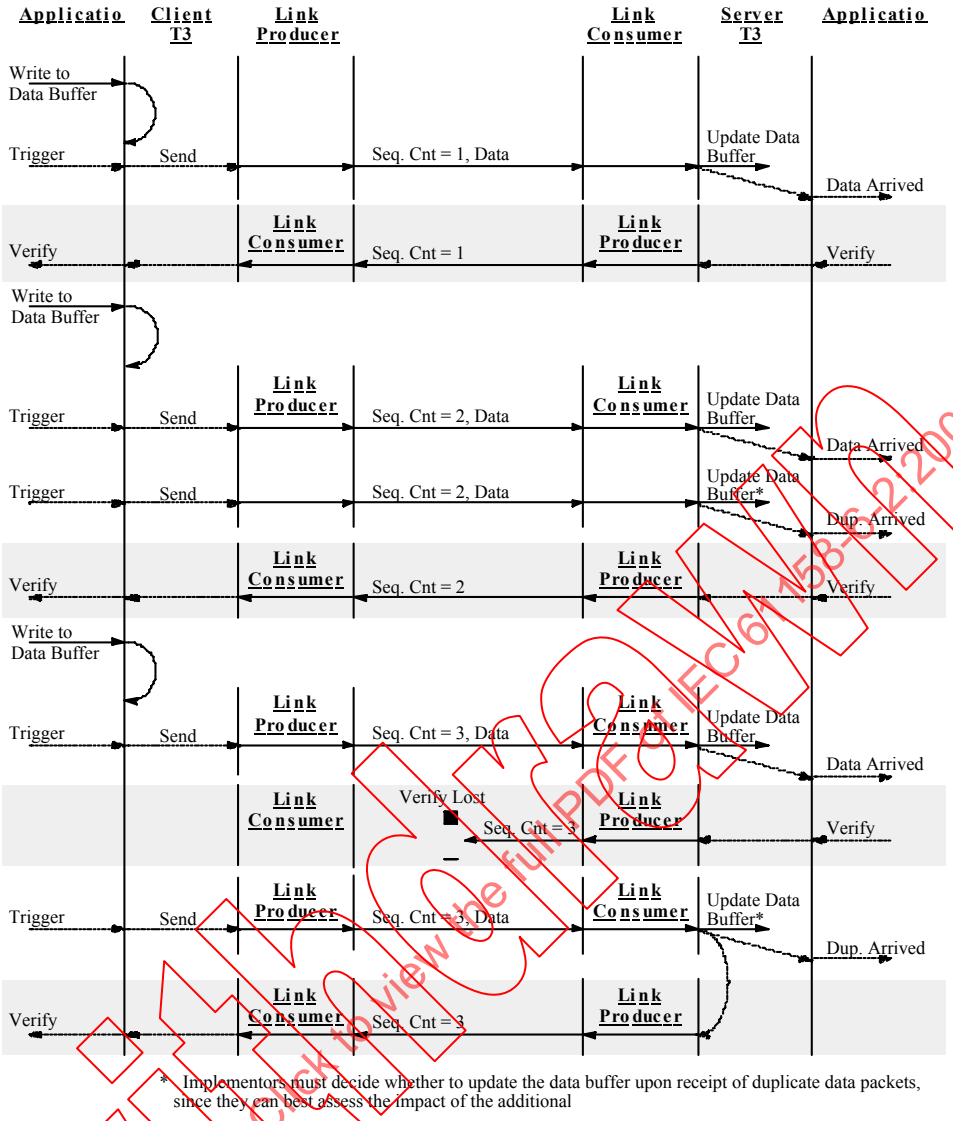


NOTE 1 The client and server transports each use two buffers: one for transmitting, and one for receiving.

NOTE 2 The server application can return data to the client application using the connection established for the verification.

Figure 47 – Data flow diagram using client transport class 3 and server transport class 3

Figure 48 shows the sequence of actions which take place during data transfer using client transport class 3 and server transport class 3. In the example depicted, the verification is the only data transmitted in the server–to–client direction. The unshaded areas in Figure 48 indicate client–to–server data transmission, and the shaded areas indicate server–to–client transmission.



* Implementors must decide whether to update the data buffer upon receipt of duplicate data packets, since they can best assess the impact of the additional

Figure 48 – Sequence diagram of data transfer using client transport class 3 and server transport class 3 *without* returned data

At times, the server–to–client transmission comprises the verification *and* additional data. Figure 49 shows the sequence in which actions take place during data transfer using client transport class 3 and server transport class 3 when the verification and data are returned to the client.

NOTE The returned data is written asynchronously to the receipt of data from the client.

The unshaded areas in Figure 49 indicate client–to–server data transmission, and the shaded areas indicate server–to–client transmission.

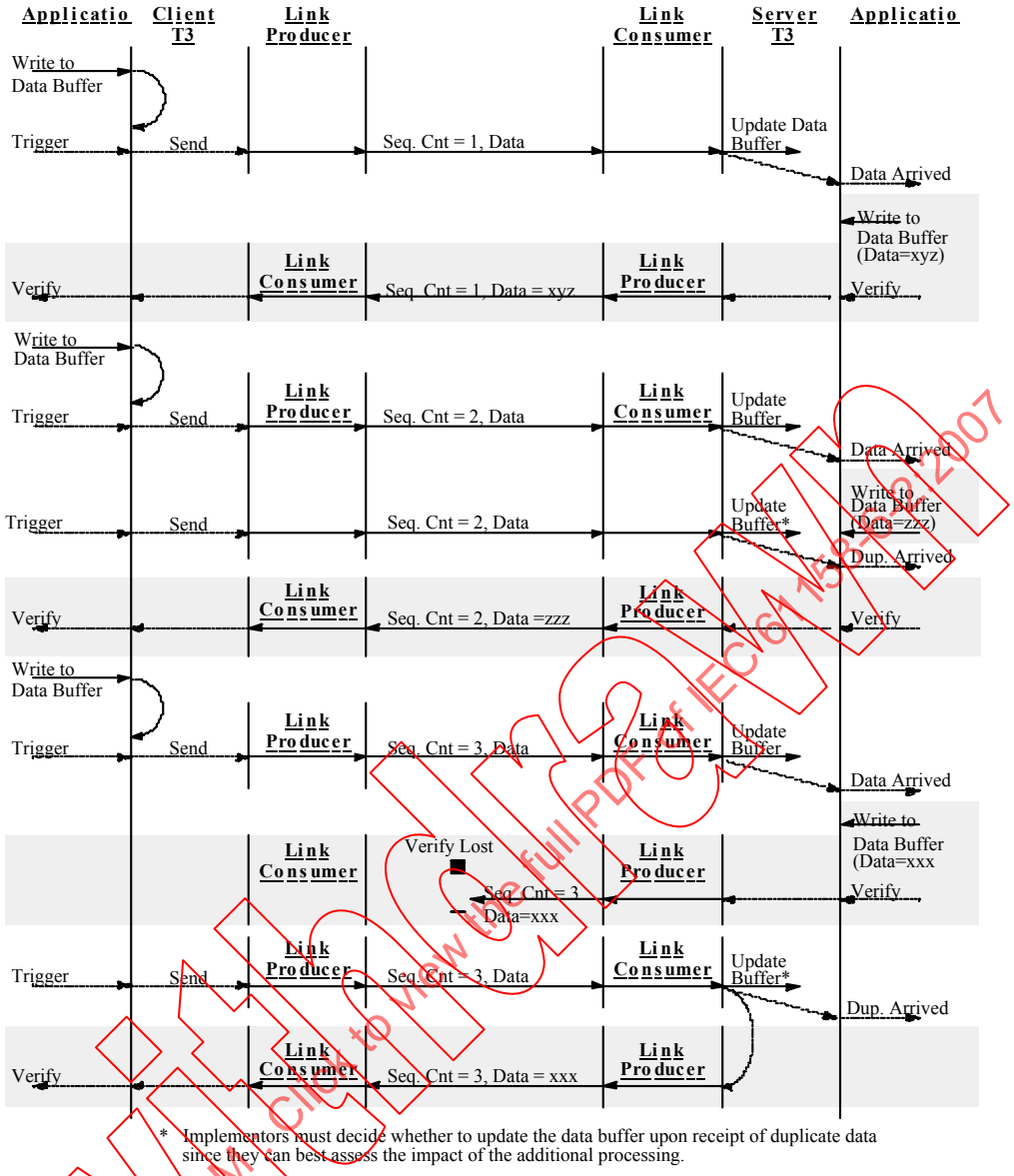


Figure 49 – Sequence diagram of data transfer using client transport class 3 and server transport class 3 *with* returned data

Possible uses of transport class 3 include change–of–state inputs and outputs, asynchronous [independent] block transfers, event acknowledgements, communication between controllers and operator interface devices, uploads, downloads, and messaging.

9.3.8.2 Transport class 3 client

9.3.8.2.1 Functions

In the client–to–server direction, a class 3 client transport is responsible for:

- accepting the client application's trigger that it has written a data packet to the client–side TPDU transmit buffer;
- writing a transport header to the client–side TPDU transmit buffer for each packet that the client application writes to it;
- initialising the sequence count in the transport header (for the first packet transmitted) or incrementing the sequence count (for subsequent packets of the same transmission);

- triggering the producing node of the network connection to produce the TPDU packet on the link and send it to the consuming node of the network connection.

In the server–to–client direction, a class 3 client transport is responsible for:

- accepting notification from the consuming node of the return network connection that it has written data (the server transport instance's verification) to the client–side TPDU receive buffer;
- locating and reading the transport header of each data packet that the consuming node of the return network connection writes to the client–side TPDU receive buffer;
- lotifying the client application that the consuming node of the return network connection has written data to the client–side TPDU receive buffer.

The application can update the data buffer or retrigger data before all of the verifications have been received. If the client transport is retriggered while waiting for verification, the previous verification received register is cleared. This means that the client transport class shall wait for all active consumers to return verification before notifying the application.

9.3.8.2.2 States

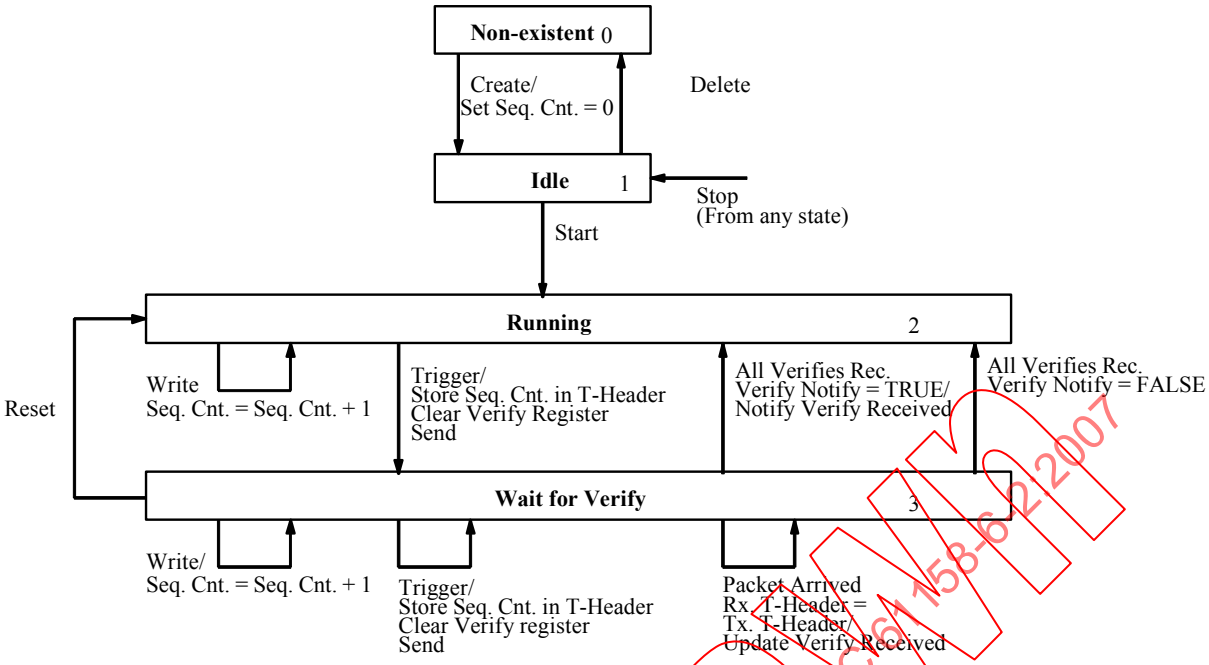
The defined states and their descriptions of the Class 3 Transport Client are listed in Table 226.

Table 226 – Class 3 transport client states

State	Description
Non-existent	The instance of Transport Class 3 does not exist.
Idle	The instance of Transport Class 3 has been created but not yet started.
Running	The instance of Transport Class 3 has been created and started.
Wait to verify	The instance of Transport Class 3 has been created and started, verification is pending.

9.3.8.2.3 State transitions

State transitions of the class 3 client transport are shown in Figure 50.



NOTE The sequence count is set to 0 by the create service. It is then incremented after every *write* event until it reaches a maximum value of 2^{16}-1 . After that it rolls over to 0.

Figure 50 – Class 3 client STD

9.3.8.2.4 State event matrix

State transitions of the class 3 client transport are shown in Table 227.

Table 227 – Class 3 client SEM

Event	State			
	Non-existent	**Idle**	**Running**	**Wait for verify**
Create	1) Seq. Cnt. = 0 2) Transition to **Idle**	Error	Error	Error
Delete	Error	Transition to **Non-existent**	Error	Error
Start	Error	Transition to **Running**	Error	Error
Stop	Error	No action	Transition to **Idle**	Transition to **Idle**
Reset	Error	Error	No Action	Transition to **Running**
Write	Error	No action	Seq. Cnt. = Seq. Cnt. + 1	Seq. Cnt. = Seq. Cnt. + 1
Trigger	Error	No action	1) Store Seq. Cnt. in TPDU 2) Clear Verify register 3) Send 4) Transition to **Wait for Verify**	1) Store Seq. Cnt. in TPDU 2) Clear Verify register 3) Send
Packet Arrived Rx. T- header = Tx. T-Header	No action	No action	No Action	1) Update Verify Register
All Verifies Received Notification = TRUE	No action	No action	Error	1) Notify: Verify 2) Transition to **Running**
All Verifies Received Notification = FALSE	No action	No action	Error	1) Transition to **Running**

NOTE The STD and SEM for client classes 2 and 3 are nearly identical. Acknowledgement has been substituted for verification.

9.3.8.3 Transport class 3 server

9.3.8.3.1 Functions

Transport class 3 starts with the behaviour in class 1 and adds a return connection that verifies that the application has received the packet. It is the responsibility of the application to verify receipt of data. The sequence count that shall be returned with the verification is the same as the sequence count sent with the data.

The application is required to read the TPDU buffer before invoking the verify service.

In the client–to–server direction, a class 3 server transport is responsible for:

- accepting the notification from the consuming node of the network connection that it has written a data packet to the server–side TPDU receive buffer;
- locating and reading the transport header of each packet that the consuming node of the network connection writes to the server–side TPDU receive buffer;
- comparing the sequence count of each packet in the server–side TPDU receive buffer with that of the previous packet;
- notifying the server application that the most recently arrived packet is a duplicate of the previous one when their sequence counts match;
- notifying the server application that data has been written to the server–side TPDU receive buffer.

In the server–to–client direction, a class 3 server transport is responsible for:

- writing the transport header of the data in the server–side TPDU receive buffer to the server–side TPDU transmit buffer;
- correlating that transport header with any data that the server application wants to send to the client application and has written to the server–side TPDU transmit buffer;
- triggering the producing node of the return network connection to produce the TPDU packet on the link and send it to the consuming node of the return network connection.

NOTE Data from the application can be return from the server to the client along the verification connection.

The send service is invoked when verify is received from an application or when a duplicate transmission is received while in the running state. Retransmitting when a previously verified packet has been received allows this class to recover from lost verifications.

9.3.8.3.2 States

The defined states and their descriptions of the Class 3 Transport Server are listed in Table 228.

Table 228 – Class 3 transport server states

State	Description
Non-existent	The instance of Transport Class 3 does not exist.
Idle	The instance of Transport Class 3 has been created but not yet started.
Read to run	The instance of Transport Class 3 has been created and started, waiting for the first packet to arrive.
Wait for verify	The instance of Transport Class 3 has been created and started, the first packet arrived, waiting for verification to be requested by the application.
Running	The instance of Transport Class 3 has been created and started, verification has been processed.

9.3.8.3.3 State transitions

State transitions of the class 3 server transport are shown in Figure 51.

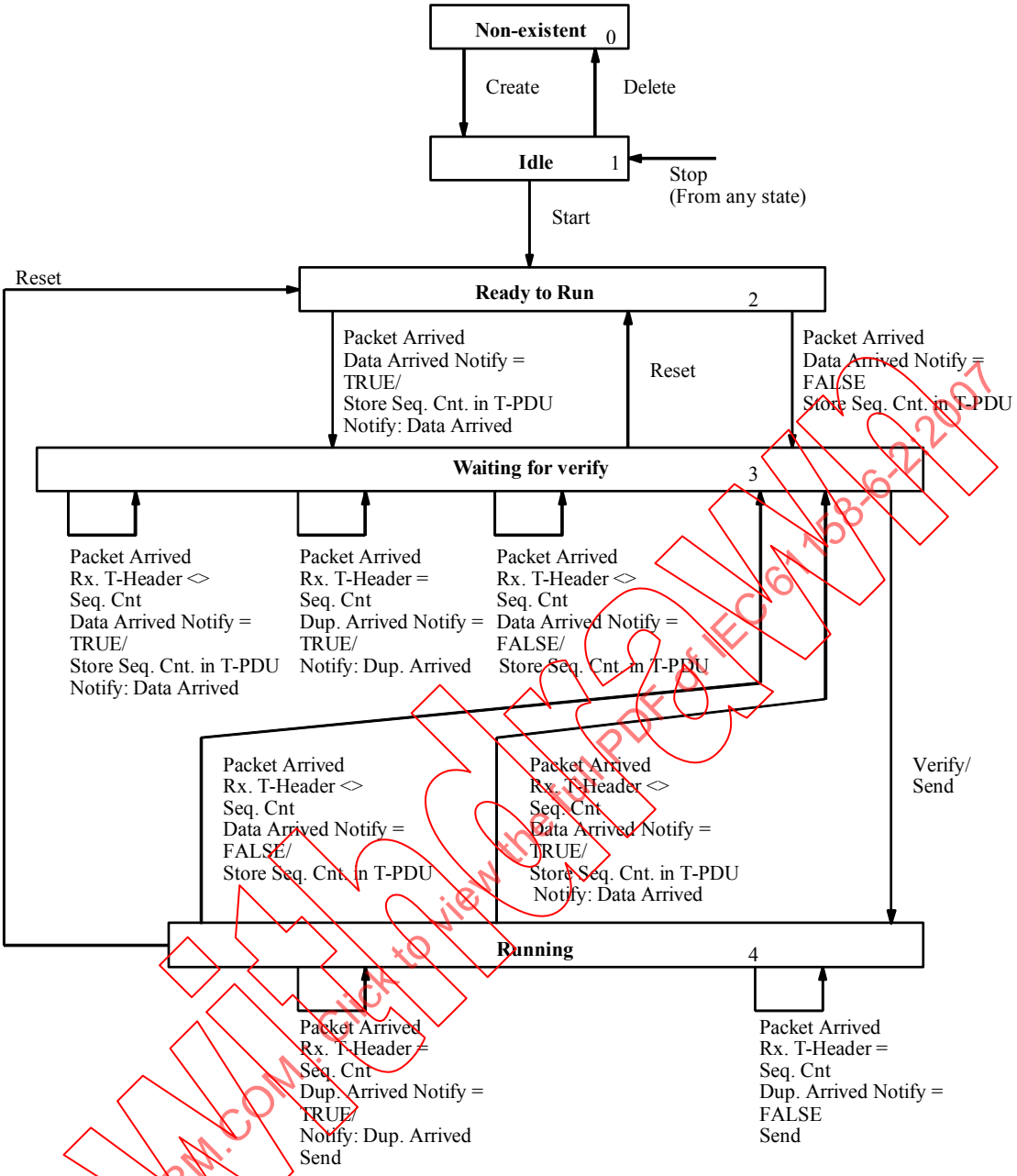


Figure 51 – Class 3 server STD

9.3.8.3.4 State event matrix

State transitions of the class 3 server transport are shown in Table 229.

Table 229 – Class 3 server SEM

Event	State				
	Non-existent	Idle	Ready to run	Waiting for verify	Running
Create	Transition to Idle	Error	Error	Error	Error
Delete	Error	Transition to Non-existent	Error	Error	Error
Start	Error	Transition to Ready to Run	Error	Error	Error
Stop	Error	No action	Transition to Idle	Transition to Idle	Transition to Idle
Reset	Error	Error	No action	Transition to Ready to Run	Transition to Ready to Run
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = TRUE Dup Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify
Packet Arrived Rx. T-Header = Seq.Count Data Arrived Notify = TRUE Dup Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify	1) Notify: Dup. Arrived	1) Notify: Dup. Arrived 2) Send (Implies Verify)
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = TRUE Dup Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify
Packet Arrived Rx. T-Header = Seq.Count Data Arrived Notify = TRUE Dup Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Notify: Data Arrived 3) Transition to Waiting for Verify	No action	1) Send (Implies Verify)
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = FALSE Dup Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify	1) Store Seq. Cnt. in TPDU	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify
Packet Arrived Rx. T-Header = Seq.Count Data Arrived Notify = FALSE Dup Arrived Notify = TRUE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify	1) Notify: Dup. Arrived	1) Notify: Dup. Arrived 2) Send (Implies Verify)
Packet Arrived Rx. T-Header <> Seq.Count Data Arrived Notify = FALSE Dup Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify	1) Store Seq. Cnt. in TPDU	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify
Packet Arrived Rx. T-Header = Seq.Count Data Arrived Notify = FALSE Dup Arrived Notify = FALSE	No action	No action	1) Store Seq. Cnt. in TPDU 2) Transition to Waiting for Verify	No action	1) Send (Implies Verify)
Verify	Error	No action	No action	1) Send (Implies Verify) 2) Transition to Running	No action

9.3.9 Transport state machines – classes 4, 5, 6

9.3.9.1 General

This subclause explains the common elements of the transport classes 4 – 6, which provide higher-level features such as non-blocking, fragmentation/assembly, and transport-level acknowledgement. Transport classes 4 and 5 shall not be used with messaging. Messaging does not provide a way to match requests with responses. Use class 3 for messaging without fragmentation.

The fragmentation protocols assume that per-connection send order is preserved throughout the system; overwrite is permissible, but changing the order of the messages is not.

9.3.9.2 Class 4 and 5 data flow diagram

Figure 52 shows the relationship of the class 4 and 5 Transport to the Application and the Link Producer and Link Consumer, the data that move between these functions, and the events which are generated and used by them. (The class 6 data flow diagram is included with the class 6 definition, see 9.3.12.2.) There are some significant differences between these and the class 3 transport.

- The application data is entered into a queue, not placed directly into the TPDU Buffer. This allows the application to give several requests to the transport without waiting for the acknowledgement, verification, or response to a previous request.
- The success or failure of an application request is placed into a queue for the application to access, rather than using a simple event. This is done because the execution of the application and the delivery of the application data are asynchronous.
- The events and data on the right hand side and the left hand side are identical. There is not a “Client” and “Server” transport for these classes. Both sides are capable of sending and receiving application data.
- The trigger timer is explicitly identified, rather than being a portion of the application. This is because the transport is using the trigger signal to perform timing functions.

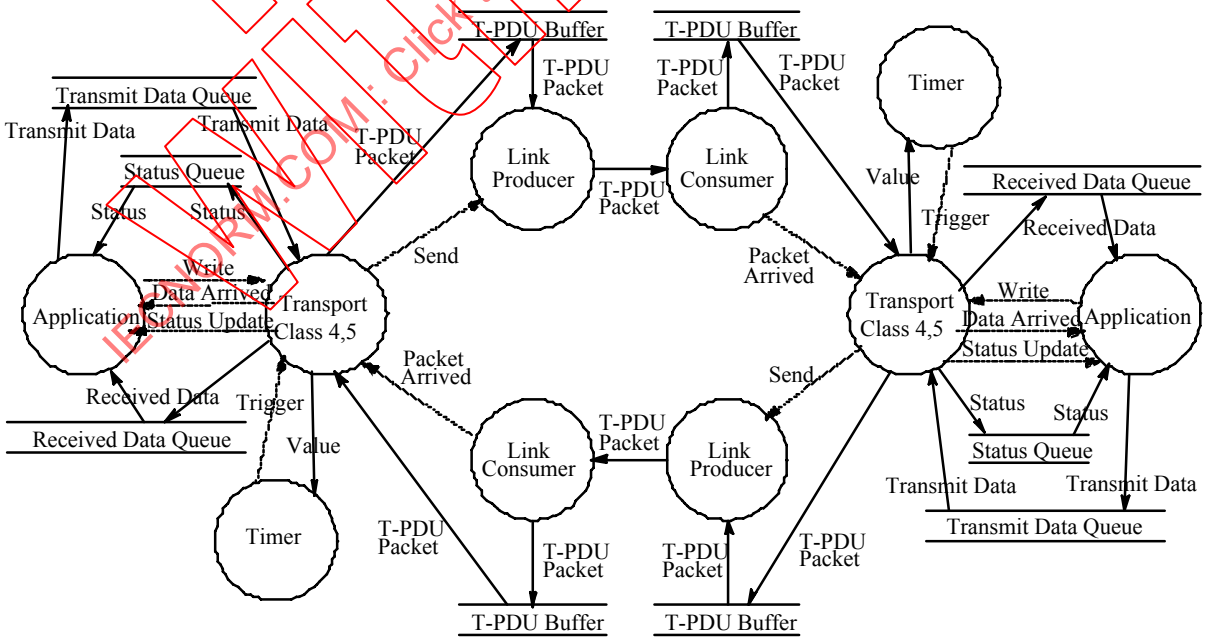


Figure 52 – Data flow diagram using transport classes 4 and 5

The Write event is used to notify the transport that new application data has been placed in the queue. The Trigger event is assumed to be provided by a resettable retry timer. These

events are used in a different sense than they are used for classes 0 to 3. These are shown in Table 230.

Table 230 – Write and trigger events in class 4 and 5 transport

	Classes 0 to 3	Classes 4 to 6
Write event	"New data available": Causes sequence count to increment.	"New data available": Indicates queue is non-empty. If the transport is not otherwise busy, shall cause data to be sent.
Trigger event	Causes data to be sent.	Causes retry or keep alive to be sent.

9.3.9.3 Classes 4 and 5 sequence diagrams

9.3.9.3.1 Typical sequences

Some typical sequences are illustrated to aid understanding of the operation of these transport classes. Formal specifications of these transport classes are given in IEC 61158-5-2.

9.3.9.3.2 Typical message exchange sequence

Figure 53 shows the sequence in which actions take place during a messaging request/response exchange using transport classes 4 and 5. The unshaded areas in Figure 53 indicate data transmission in one direction, and the shaded areas indicate data transmission in the other direction. Each transmission shows the Command and Sequence Number used for each header (SND and RCV). SND gives the command in the direction of the data flow, and RCV is the acknowledgement of the command in the other direction.

When a transport has some data to send, it uses the *Only* command and when it has nothing to send, it uses the *Null* command. It is recommended that the Sender state machine send zero application data octets with the *Null* command. It is also recommended that the Receiver state machine accept *Null* commands with a non-zero number of application data octets, which it ignores.

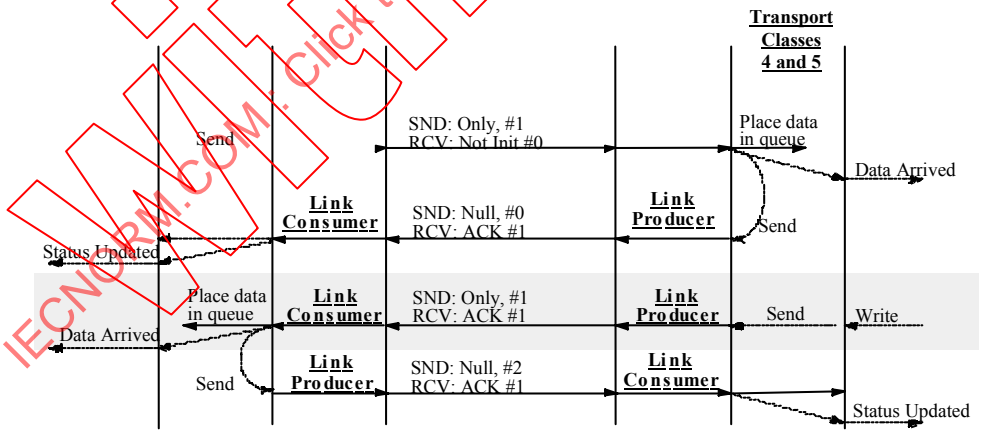


Figure 53 – Sequence diagram of message exchange using transport classes 4 and 5

9.3.9.3.3 Typical sequence with overwrite

It is permissible for a transport to send two data packets in a row. This implies that the second one might overwrite the first one in an intermediate module. This is desirable, since the second packet contains information that is newer than the first one and the data in the first one is no longer needed.

In the previous example (see 9.3.9.3.2), the transport on the right had nothing to send when it received the *Only* #1 from the transport on the left. It needed to send the *Null* #0, so that it

could deliver the ACK #1. But when the application on the right did a WRITE, the transport immediately sent the *Only* #1 along with the ACK #1. The previous example shows both of them arriving separately. Figure 54 shows the left side's *Only* #1 overwriting the *Null* #0.

Figure 54 also shows a different situation where an overwrite can occur. The right side sends a *Null* #2 with the ACK #2 as the result of the keep-alive *Trigger* event. But, before it can be delivered, the right side receives the *Only* #3 from the left side. The left side immediately sends the *Null* #2, but this time with the newer ACK #3, which overwrites the ACK #2.

It is not necessary to wait for the ACK to a *Null* before sending an *Only*.

When sending an *Only*, it shall have a different sequence number than the *Null*, so sender knows which is being ACK'd by the other side.

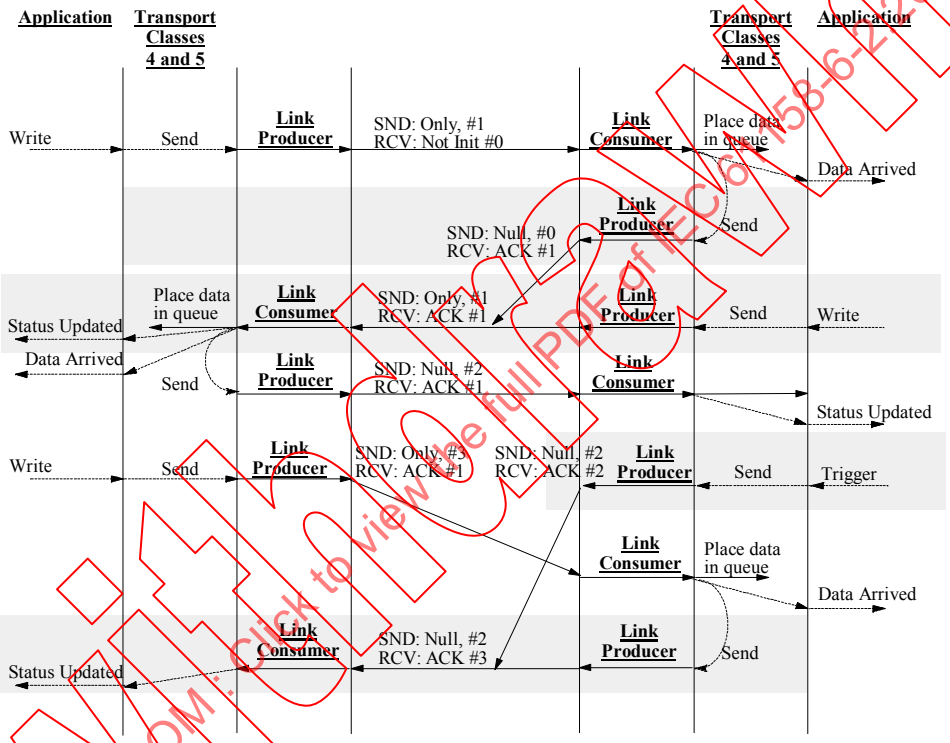


Figure 54 – Sequence diagram of messages overwriting each other

9.3.9.3.4 Typical queued sequence

Figure 55 shows the sequence in which actions take place during a messaging request/response exchange using transport classes 4 and 5. In this case, the applications place some data into the send queue before the first data transfer has been acknowledged. The unshaded areas in Figure 55 indicate data transmission in one direction, and the shaded areas indicate data transmission in the other direction.

In this case, the left transport shall not use the *Null* #2 to deliver the ACK #1, but shall send the data from its queue via the *Only* #2.

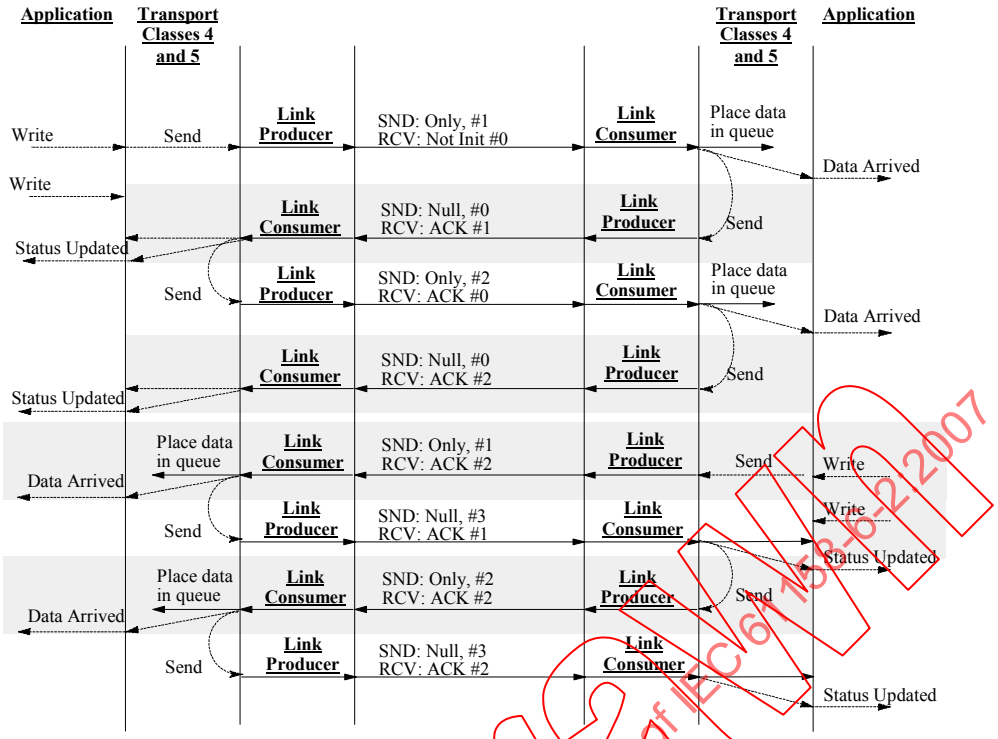


Figure 55 – Sequence diagram of queued message exchange using transport classes 4 and 5

9.3.9.3.5 Typical sequence with retries

Retries are performed when a transport does not receive a Receiver header with the same sequence count as its current Sender header in the timeout period. The TPDU is not changed for a retry. Figure 56 shows two retry situations: where an *Only* command fails to be delivered and where a *Null* with an expected ACK fails to be delivered. The Trigger event is generated by the retry timer.

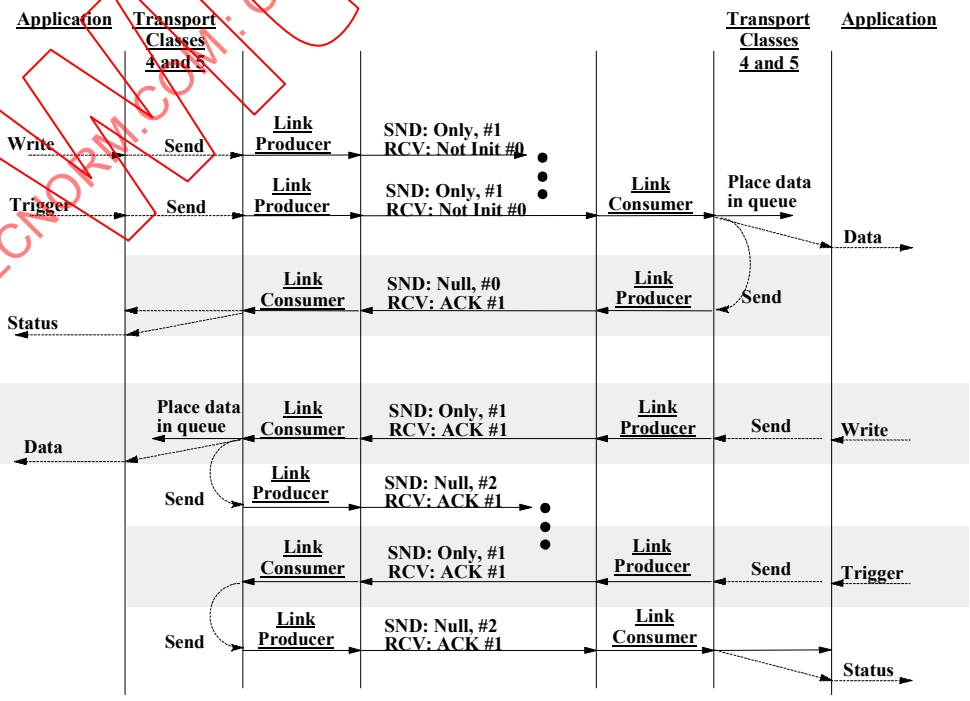


Figure 56 – Sequence diagram of retries using transport classes 4 and 5

9.3.9.3.6 Typical idle sequence

When a class 4 or 5 transport has nothing to send which shall impact the other state machine, it does not need to send anything. However, it is necessary to send some small amount of traffic to keep the connection alive. The Trigger event shall be used, with the timer set to a "keep alive" value. Figure 57 shows the *Null* command used to keep the connection alive, using the Trigger event.

NOTE the *Null* command does not need to be acknowledged. The other transport shall not retry a *Null* command. It shall continue to resend the *Null* command at the "keep alive" rate.

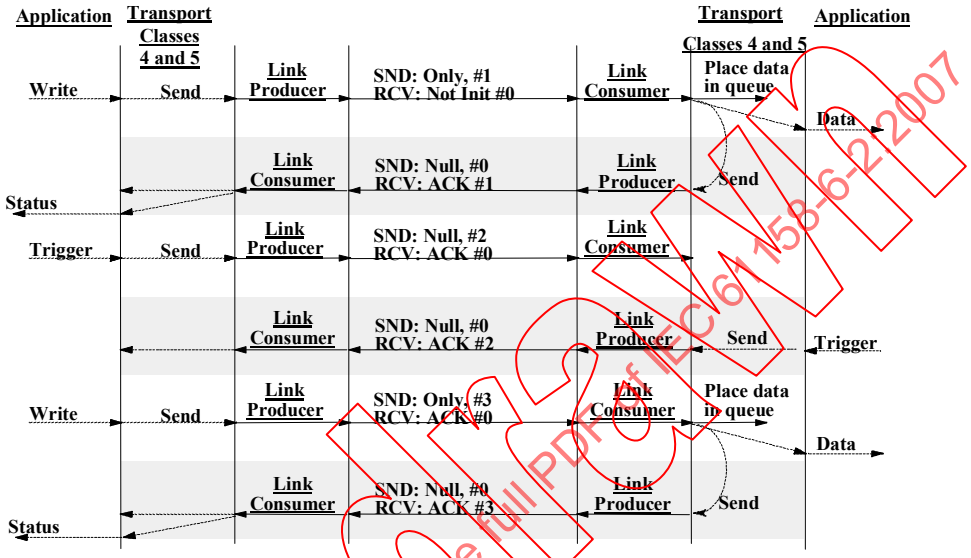


Figure 57 – Sequence diagram of idle traffic using transport classes 4 and 5

9.3.9.4 Classes 4 to 6 basic structure

To make the specification of the Non-Blocking classes (Classes 4 and 5) easier to read, the specification describes them as being made up of two state machines: one to handle the "Sender" function (Transport PDU's from this device to the other) and the other to handle the "Receiver" function (Transport PDU's from the other device to this one). (The class 6 transport has a client and a server state machine similar to the class 3 transport.)

The Sender and Receiver state machines in one node are coupled to each other via one event and three data items. The event and the data items come from the Receiver state machine to the Sender one. The *NewData* event indicates to the Sender state machine that one or more of the data items have changed. The *To_Send* and *From_Recv* data items are copies of the headers as shown in the diagram. The *Flag* data item is used by the Receiver state machine to tell the Sender state machine that it shall generate a Send event to the link producer, even if the Sender one has no reason to do so. This usually is accompanied by a change in the *From_Recv* data item, but not all changes in *From_Recv* need to be sent out immediately.

The Sender state machine has control of the Link Producer; the Receiver state machine accepts the data from the Link Consumer. The Sender state machine accepts data from the application; the Receiver state machine delivers data to the application. (This application data might be either Application Requests or Responses or data; the transport does not care nor know which it is.)

Figure 58 shows the relationship of these state machines. Both state machines are contained in one end node. The other end is identical.

Figure 59 shows the basic structure for class 6.

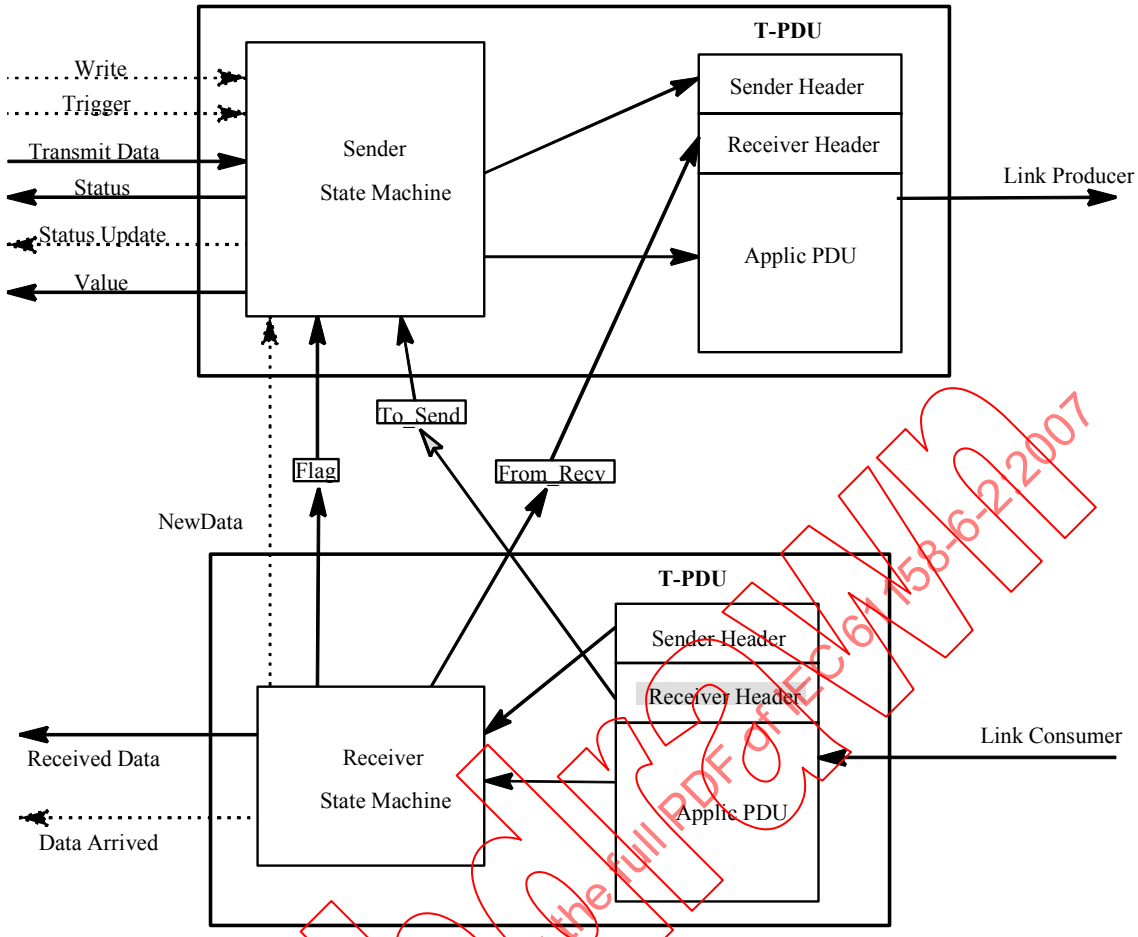


Figure 58 – Classes 4 and 5 basic structure

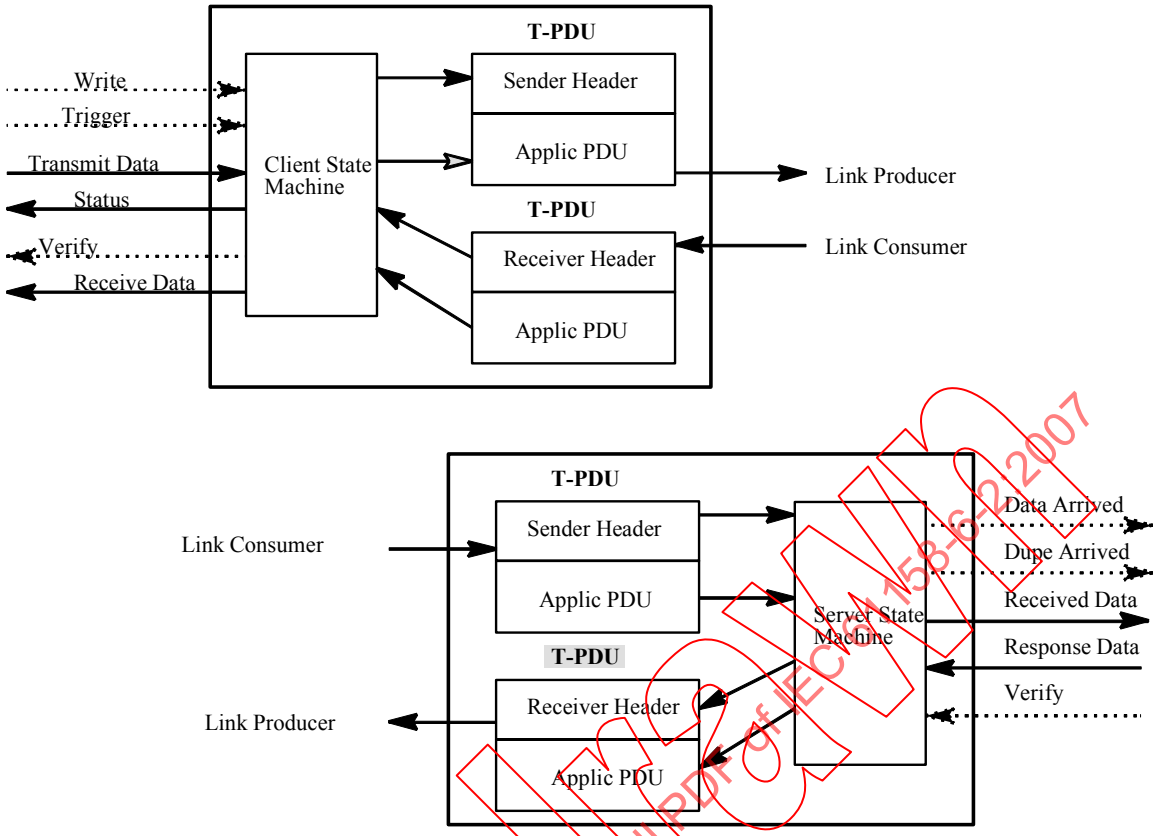


Figure 59 – Class 6 basic structure

9.3.9.5 Classes 4 to 6 general states

The defined states and their descriptions of the Class 4 Transport Sender are listed in Table 231.

Table 231 – Common states for transport classes 4 to 6

State	Description
Non-existent	The instance of Transport Class 4 to 6 does not exist.
Idle	The instance of Transport Class 4 to 6 has been created but not yet started.
Operational	The instance of Transport Class 4 to 6 has been created and started. NOTE This state has sub-states described in features particular to individual classes 4, 5 and 6.

9.3.9.6 Classes 4 to 6 general state transitions

Figure 60 shows the general behaviour of the transport classes. Each of the classes has some additional details added to the “Operational” state. This general behaviour shall **not** be repeated for each class in the interest of emphasizing the common behaviours and allowing those other descriptions to focus on the unique aspects of that class.