INTERNATIONAL STANDARD

ISO 17356-4

First edition
2005-11-01

Road vehicles — Open interface for embedded automotive applications —

Part 4: OSEK/VDX Communication (COM)

Véhicules routiers — Interface ouverte pour applications automobiles embarquées —

Partie 4: Communications (COM) OSEK/VDX

Reference number
ISO 17356-4:2005(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

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

ISO 17356-4 was prepared by Technical Committee ISO/TC 22, *Road vehicles*, Subcommittee SC 3, *Electrical and electronic equipment*.

ISO 17356 consists of the following parts, under the general title *Road vehicles — Open interface for embedded automotive applications*:

— *Part 1: General structure and terms, definitions and abbreviated terms*

— *Part 2: OSEK/VDX specifications for binding OS, COM and NM*

— *Part 3: OSEK/VDX Operating System (OS)*

— *Part 4: OSEK/VDX Communication (COM)*

— *Part 5: OSEK/VDX Network Management (NM)*

— *Part 6: OSEK/VDX Implementation Language (OIL)*

Introduction

This part of ISO 17356 specifies a uniform communication environment for automotive control unit application software. It increases the portability of application software modules by defining common software communication interfaces and behaviour for internal communication [communication within an electronic control unit (ECU)] and external communication (communication between networked vehicle nodes), which is independent of the communication protocol used.

This part of ISO 17356 describes the behaviour within one ECU. It assumes that the communication environment described in this part of ISO 17356 is used together with an operating system that conforms to ISO 17356-3. For information on how to run the communication environment described in this part of ISO 17356 on operating systems that do not conform to ISO 17356-3, refer to Annex A.

Requirements

The following main requirements are fulfilled by this part of ISO 17356:

General communication functionality

This part of ISO 17356 offers services to transfer data between tasks and/or interrupt service routines. Different tasks may reside in one and the same ECU (internal communication) or in different ECUs (external communication). Access to ISO 17356-4 services is only possible via the specified Application Program Interface (API).

Portability, reusability and interoperability of application software

It is the aim of this part of ISO 17356 to support the portability, reusability and interoperability of application software. The API hides the differences between internal and external communication as well as different communication protocols, bus systems and networks.

Scalability

This part of ISO 17356 ensures that an ISO 17356-4 implementation can run on many hardware platforms. The implementation requires only a minimum of hardware resources, therefore different levels of functionality (conformance classes) are provided.

Support for ISO 17356-5 (Network Management-NM):

Services to support Indirect NM are provided. Direct NM has no requirements of this part of ISO 17356.

Communication concept

Figure 1 shows the conceptual model of this part of ISO 17356 and its positioning within the architecture defined by ISO 17356. This model is presented for better understanding, but does not imply a particular implementation of this part of ISO 17356.

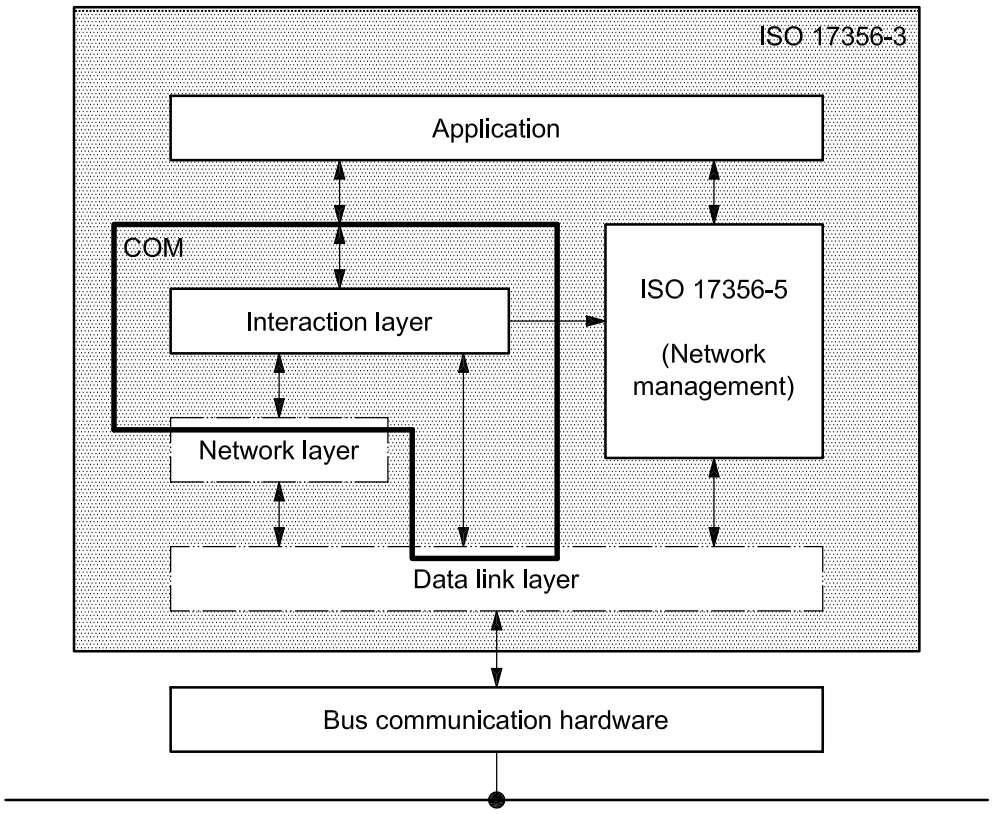


Figure 1 — COM's layer model

In this model, the scope of this part of ISO 17356 partly or entirely covers the following layers:

Interaction Layer

The Interaction Layer (IL) provides the ISO 17356-4 API which contains services for the transfer (send and receive operations) of messages. For external communication it uses services provided by the lower layers, whereas internal communication is handled entirely by the IL.

Network Layer

The Network Layer handles — depending on the communication protocol used — message segmentation/recombination and acknowledgement. It provides flow control mechanisms to enable the interfacing of communication peers featuring different levels of performance and capabilities. The Network Layer uses services provided by the Data Link Layer. This part of ISO 17356 does not specify the Network Layer; it merely defines minimum requirements for the Network Layer to support all features of the IL.

Data Link Layer

The Data Link Layer provides the upper layers with services for the unacknowledged transfer of individual data packets (frames) over a network. Additionally, it provides services for the NM. This part of ISO 17356 does not specify the Data Link Layer; it merely defines minimum requirements for the Data Link Layer to support all features of the IL.

Structure of this document

In the following text, the specification chapters are described briefly. Clauses 1 to 5 are normative, the appendices are descriptive.

Clause 1: Scope
This clause describes the motivation and requirements for this part of ISO 17356, the conceptual model used and the structure of the document.

Clause 2: Normative references

Clause 3: Interaction Layer

This clause describes the functionality of the IL of the ISO 17356-4 model and defines its API.

Clause 4: Minimum requirements of lower communication layers

This clause lists the requirements imposed by this part of ISO 17356 on the lower communication layers (Network Layer and Data Link Layer) to support all features of the IL.

Clause 5: Conformance Classes

This clause specifies the Communication Conformance Classes, which allow the adaptation of the feature content of ISO 17356-4 implementations to the target system's requirements.

Annex A: Use of this part of ISO 17356 (Com) with an OS not conforming to ISO 17356-3

Annex A gives hints on how to run this part of ISO 17356 on operating systems that do not conform to ISO 17356-3.

Annex B: Application notes

Annex B provides information on how to meet specific application requirements with the given ISO 17356-4 model.

Annex C: Callouts

Annex C supplies application examples for callouts.

Road vehicles — Open interface for embedded automotive applications —

Part 4: OSEK/VDX Communication (COM)

1 Scope

This part of ISO 17356-4 (COM) specifies a uniform communication environment for automatic control unit application software.

It increases the portability of application software modules by defining common software communication interfaces and behaviours for internal communication (communication within an ECU) and external communication (communication between networked vehicle nodes), which is independent of the used communication protocol.

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.

ISO 17356-2, *Road vehicles — Open interface for embedded automotive applications — Part 2 OSEK/VDX specifications for binding OS, COM and NM*

ISO 17356-3, *Road vehicles — Open interface for embedded automotive applications — Part 3 OSEK/VDX Operating System (OS)*

ISO 17356-5, *Road vehicles — Open interface for embedded automotive applications — Part 5 OSEK/VDX Network Management (NM)*

ISO 17356-6, *Road vehicles — Open interface for embedded automotive applications — Part 6 OSEK/VDX Implementation Language (OIL)*

3 Interaction Layer

3.1 Overview

3.1.1 Presentation

The communication in this part of ISO 17356 is based on messages[1)]. A message contains application-specific data. Messages and message properties are configured statically via OIL (ISO 17356-6). The content and usage of messages is not relevant to this part of ISO 17356. Messages with a length of zero (see *zero-length messages*, Annex B) are allowed.

1) Messages are often called *signals*. Thus, COM offers a signal-based interface.

In the case of internal communication, the Interaction Layer (IL) makes the message data immediately available to the receiver (see Figure 2). In the case of external communication, the IL packs **one or more** messages into assigned *Interaction Layer Protocol Data Units* (I-PDU) and passes them to the underlying layer. The functionality of internal communication is a subset of the functionality of external communication. Internal-external communication occurs when the same message is sent internally as well as externally.

Administration of messages is done in the IL based on *message objects*. Message objects exist on the sending side (sending message object) and on the receiving side (receiving message object).

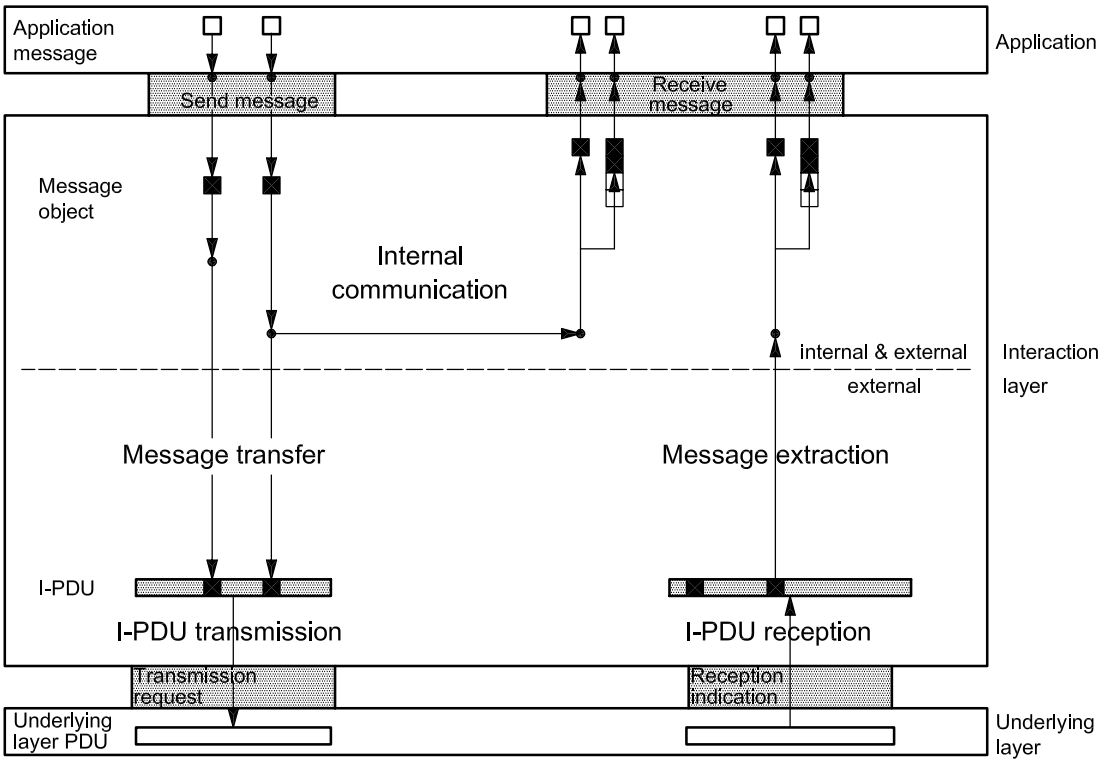


Figure 2 — Simplified model for message transmission and reception in ISO 17356-4

The data that is communicated between the IL and the underlying layer is organized into I-PDUs which contain one or more messages (see Figure 2). A message shall occupy contiguous bits within an I-PDU and shall not be split across I-PDUs. Within an I-PDU messages are bit-aligned. The size of a message is specified in bits.

The byte order (endianess) in a CPU can differ from the network representation or from other CPUs on the network. Therefore, to provide interoperability across the network, the IL provides a conversion from the network representation to the local CPU representation and vice versa, which is statically configured on a per-message basis.

The IL offers an Application Program Interface (API) to handle messages. The API provides services for initialization, data transfer and communication management. Services transmitting messages over network are non-blocking. This implies, for example, that a service that sends a message is unable to return a final transmission status because the transfer to the network is still in progress. This part of ISO 17356 provides *notification mechanisms* for an application to determine the status of a transmission or reception.

The functionality of the IL can be extended by *callouts.* (3.8 contains a description of where callouts can be inserted.)

3.1.2 Communication concept

Senders and receivers of messages are either tasks or interrupt service routines (ISRs) in an OS. Messages are sent to sending message objects and received from receiving message objects.

Message objects are identified using message identifiers. Message identifiers are assigned to message objects at system generation.

This part of ISO 17356 supports communication from “m” senders to “n” receivers (m:n communication). Zero or more senders can send messages to the same sending message object. Sending message objects are configured to store messages in zero or more receiving message objects for internal communication and in zero or one I-PDUs for external communication.

One or more sending message objects can be configured to store messages in the same I-PDU for external communication.

An I-PDU can be received by zero or more CPUs. In each CPU which receives the I-PDU, each message contained in the I-PDU is stored in zero or more receiving message objects. Zero or more receivers can receive messages from a receiving message object (see Annex B for additional information).

A receiving message object receives messages from either exactly one sending message object (internal communication) or exactly one I-PDU, or it receives no messages at all.

A receiving message object can be defined as either queued or unqueued. While a message received by a message object with the property “queued” (queued message) can only be read once (the read operation removes the oldest message from the queue), a message received from a message object with the property “unqueued” (unqueued message) can be read more than once; it returns the last received value each time it is read.

The queue size for message objects with the property “queued” is specified per message object and shall not be zero. If the queue of a receiving message object is full and a new message arrives, this message is lost.

This part of ISO 17356 is not responsible for allocating memory for the application messages, but it allows independent access to message objects for each sender and receiver. In the case of unqueued messages, an arbitrary number of receivers may receive the message. In the case of queued messages, only one receiver may receive the message. The IL guarantees that the data in the application’s message copies are consistent by the following means: the IL deals with messages automatically, and application message data is only read or written during a send or receive service call.

An external message can have one of two transfer properties:

— Triggered Transfer Property: the message in the assigned I-PDU is updated and a request for the I-PDU’s transmission is made.

— Pending Transfer Property: the message in the I-PDU is updated without a transmission request.

Internal messages do not have a transfer property. They are immediately routed to the receiver side.

There are three transmission modes for I-PDUs:

— Direct Transmission Mode: the transmission is explicitly initiated by sending a message with Triggered Transfer Property.

— Periodic Transmission Mode: the I-PDU is transmitted repeatedly with a pre-set period.

— Mixed Transmission Mode: the I-PDU is transmitted using a combination of both the Direct and the Periodic Transmission Modes.

This part of ISO 17356 supports only static message addressing. A statically addressed message has zero or more receivers defined at system generation time, each of which receives the message whenever it is sent. A message has either a static length or its length may vary up to some statically defined maximum. Messages with a maximum length are called *dynamic-length messages*.

This part of ISO 17356 provides a mechanism for monitoring the transmission and reception timing of messages, called *Deadline Monitoring*. Deadline Monitoring verifies on the sender side that the underlying layer confirms transmission requests within a defined time period and on the receiver side that periodic messages are received within a defined time period. The monitoring is performed based on I-PDUs.

The IL provides a fixed set of *filter* algorithms. On the sender side, a filter algorithm may be used which, depending on the message contents, discards the message. In this case, no external transmission is performed and the I-PDU is not updated. There is no filtering on the sender side for internal transmission. On the receiver side, a filter mechanism may be used per receiver in both internal and external transmission. For more details on filtering see 3.2.3 and 3.3.6.

3.1.3 Configuration

The configuration of messages and of their senders and receivers shall be defined at system generation time. Messages cannot be added or deleted at run-time, nor can the packing of messages to I-PDUs be changed. This applies to all configuration elements and their attributes unless otherwise stated.

Examples for configurable items include:

— Configuration of the transfer properties of messages and the transmission modes of I-PDUs,

— Packing of the messages to I-PDUs, and

— Usage of a queue by a receiver and the size of this queue.

The configuration of single CPUs is described in ISO-17356-6.

3.2 Message reception

3.2.1 General

This subclause states the services and functionality requirements of the message reception entity of the IL.

3.2.2 Message reception overview

The first few steps described in this section are applicable for external communication only.

Reception of a message starts with an *indication* of the delivery of its containing PDU from the underlying layer. If this indication does not yield an error, the reception was successful. In this case, an *I-PDU Callout* is called (if configured) and this PDU is copied into the I-PDU.

In the case of unsuccessful PDU reception *error indication* takes place and no data is delivered to the IL. Error indication can lead to *Message Reception Error* notification (Notification Class 3, described in 3.6.2).

After copying the data into the I-PDU further processing is performed separately for each contained message. If the I-PDU contains zero-length messages, these are processed last.

The *Reception Deadline Monitoring* takes place as described in Clause 3.5.1. Deadline Monitoring can invoke *Message Reception Error* notification (Notification Class 3, described in 3.6.2) when the message reception deadline is missed because the I-PDU that contains the message is not received in time.

The message data is then unpacked from the I-PDU and, if configured, a *Network-order Message Callout* is called for the message. Message *byte order conversion* is performed to convert from network representation

to the representation on the local CPU and, if configured, a *CPU-order Message Callout* is called for the message.

The following steps are applicable for both internal and external communication.

The *filtering* is applied to the message content. If the message is not filtered out, then the message data is copied into the receiver message object.

After filtering, *Message Reception* notification (Notification Class 1, described in 3.6.2) is invoked as appropriate. Notification is performed per message object.

Message data are copied from message object to application messages when the application calls the *ReceiveMessage* or *ReceiveDynamicMessage* API services.

3.2.3 Reception filtering

Filtering provides a means to discard the received message when certain conditions, set by message filter, are not met for the message value. The message filter is a configurable function that filters messages out according to specific algorithms. For each message, a different filtering condition can be defined through a dedicated algorithm.

Filtering is only used for messages that can be interpreted as C language unsigned integer types (characters, unsigned integers and enumeration).

For zero-length messages and dynamic-length messages, no filtering takes place.

While receiving messages, only the message values allowed by the filter algorithms pass to the application. If a value has been filtered out, the current instance of the message in the IL represents the last message value that passed through the filter.

Message filtering is performed per message object.

The following attributes are used by the set of filter algorithms (see Table 1):

— *new_value*: current value of the message;

— *old_value*: last value of the message (initialized with the initial value of the message, updated with *new_value* if the new message value is not filtered out);

— *mask, x, min, max, period, offset*: constant values; and

— *occurrence*: a count of the number of occurrences of this message.

If the message filter algorithm is *F_Always* for any particular message, no filter algorithm is included in the runtime system for the particular message.

Table 1 — Message filter algorithms

Algorithm reference	Algorithm	Description
F_Always	True	No filtering is performed so that the message always passes.
F_Never	False	The filter removes all messages.
F_MaskedNewEqualsX	*(new_value&mask) == x*	Pass messages whose masked value is equal to a specific value.
F_MaskedNewDiffersX	*(new_value&mask) != x*	Pass messages whose masked value is not equal to a specific value.
F_NewIsEqual	*new_value == old_value*	Pass messages which have not changed.
F_NewIsDifferent	*new_value != old_value*	Pass messages which have changed.
F_MaskedNewEqualsMaskedOld	*(new_value&mask) == (old_value&mask)*	Pass messages where the masked value has not changed.
F_ MaskedNewDiffersMaskedOld	*(new_value&mask) != (old_value&mask)*	Pass messages where the masked value has changed.
F_NewIsWithin	*min <= new_value <= max*	Pass a message if its value is within a predefined boundary.
F_NewIsOutside	*(min > new_value) || (new_value > max)*	Pass a message if its value is outside a predefined boundary.
F_NewIsGreater	*new_value > old_value*	Pass a message if its value has increased.
F_NewIsLessOrEqual	*new_value <= old_value*	Pass a message if its value has not increased.
F_NewIsLess	*new_value < old_value*	Pass a message if its value has decreased.
F_NewIsGreaterOrEqual	*new_value >= old_value*	Pass a message if its value has not decreased.
F_OneEveryN	*occurrence % period == offset*	Pass a message once every N message occurrences. Start: occurrence = 0. Each time the message is received or transmitted, occurrence is incremented by 1 after filtering. Length of occurrence is 8 bit (minimum).

3.2.4 Copying message data into message objects data area

Message data that are not filtered out are copied into the message object's data. One message may be delivered to one message object or more than one message object. In the latter case, the message objects may be a combination of any number of queued or/and unqueued messages.

Zero-length messages do not contain data. However, the notification mechanisms work in the same way as for non zero-length messages.

3.2.5 Copying data to application messages

The message object's data are copied to the application message by the API services *ReceiveMessage* or *ReceiveDynamicMessage*. The application provides the application message reference to the service.

This transfer of information between IL and application occurs for internal, external and internal-external communication.

For zero-length messages, no data is copied.

3.2.6 Unqueued and queued messages

3.2.6.1 Queued message

A queued message behaves like a FIFO (first-in first-out) queue. When the queue is empty, the IL does not provide any message data to the application. When the queue is not empty and the application receives the message, then the IL provides the application with the oldest message data and removes this message data from the queue.

If new message data arrives and the queue is not full, this new message is stored in the queue. If new message data arrives and the queue is full, this message is lost and the next *ReceiveMessage* call on this message object returns the information that a message has been lost.

NOTE For m:n communication, a separate queue is supported for each receiver and messages from these queues are consumed independently.

3.2.6.2 Unqueued message

Unqueued messages do not use the FIFO mechanism. The application does not consume the message during reception of message data – instead, a message may be read multiple times by an application once the IL has received it.

If no message has been received since the start of the IL, then the application receives the message value set at initialization.

Unqueued messages are overwritten by newly arrived messages.

3.3 Message transmission

3.3.1 Message transmission overview

Sending a message requires the transfer of the application message to the I-PDU (external communication) and/or the receiving message object(s) (internal communication).

A message that is transferred can be stored in zero or more message objects for internal receivers and in zero or one I-PDU for external communication.

The application message is transferred upon calling a specific API service (*SendMessage*, *SendDynamicMessage* or *SendZeroMessage*).

When the API service is called for internal communication, the message is directly handed to the receiving part of the IL (see 3.2) for further processing.

The following description is for external communication only.

For external communication, filtering on the sending side is performed. If the message is discarded, no further action takes place. No filtering takes place on zero-length messages or dynamic-length messages.

Thereafter, if configured, the *CPU-order Message Callout* is called, byte order conversion is performed, the *Network-order Message Callout* is called and the message is stored in the I-PDU.

The transfer of information between the application and IL may use any of the applicable transfer properties of messages: Triggered or Pending.

Transmission of messages via the underlying layers takes place based on I-PDUs. Transmission of I-PDUs may use any of the applicable transmission modes of I-PDUs: Direct, Periodic or Mixed.

More than one message may be stored in an I-PDU. However, only the last message in an I-PDU may be a dynamic-length message. Static-length messages may overlap each other, but it is not allowed for any message to overlap a dynamic-length message. Two messages are defined as overlapping if they have at least one I-PDU bit in common.

The moment when transmission is initiated, the *I-PDU Callout* is called.

The user can be notified if the I-PDU is transferred successfully (by confirmation from the underlying layer not containing an error) or not (by confirmation from the underlying layer containing an error, or by a time-out).

3.3.2 Transfer of internal messages

Internal messages do not have transfer properties because the transfer is always executed in the same way. The IL routes internal messages directly to the receiving part of the IL (see 3.2) for further processing. The application is responsible for requesting each transfer of an internal message using the *SendMessage* or *SendZeroMessage* API service.

No data transfer takes place for zero-length messages.

3.3.3 Transfer properties for external communication

3.3.3.1 Basics

This part of ISO 17356 supports two different transfer properties for the transfer of external messages from the application to the I-PDU: Triggered and Pending.

The application is responsible for requesting each transfer of a message to the IL, using the *SendMessage*, *SendDynamicMessage* or *SendZeroMessage* API services. Depending on filtering (for *SendMessage* only), the message can be discarded. If the message is not discarded, the IL stores it in the corresponding I-PDU.

No data transfer takes place for zero-length messages.

Zero-length messages can only have Triggered Transfer Property.

Even if no transmission has taken place since the last call to *SendMessage* or *SendDynamicMessage*, the I-PDU is updated.

3.3.3.2 Triggered Transfer Property

The Triggered Transfer Property causes immediate transmission of the I-PDU, except if Periodic Transmission Mode is defined for the I-PDU.

3.3.3.3 Pending Transfer Property

The Pending Transfer Property does not cause transmission of the I-PDU.

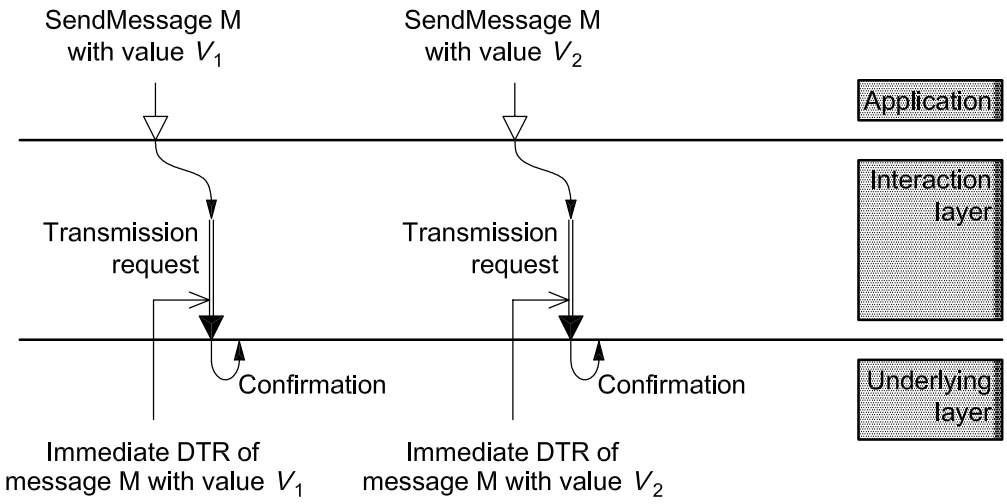
3.3.4 Transmission modes

3.3.4.1 General

This part of ISO 17356 supports three different transmission modes for the transmission of I-PDUs via the underlying layers: Direct, Periodic and Mixed.

3.3.4.2 Direct Transmission Mode

Transmission of an I-PDU with *Direct Transmission Mode* is caused by the transfer of any message assigned to the I-PDU with Triggered Transfer Property. The transfer is immediately followed by a transmission request from the IL to the underlying layer.



DTR = Direct Transmission Request

Figure 3 — Direct Transmission Mode

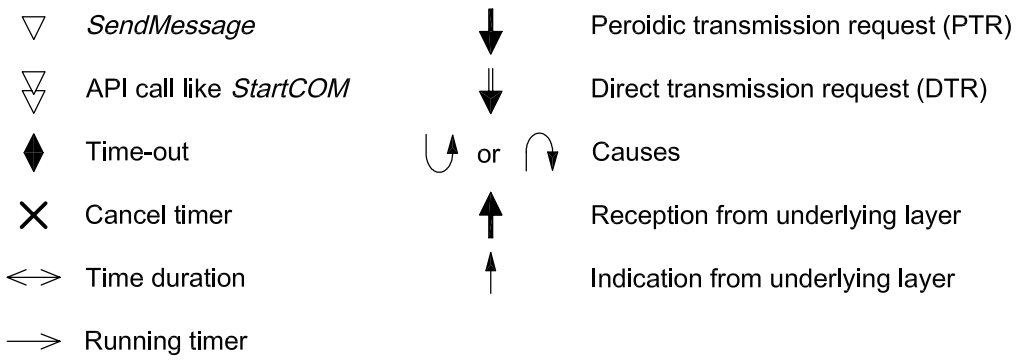
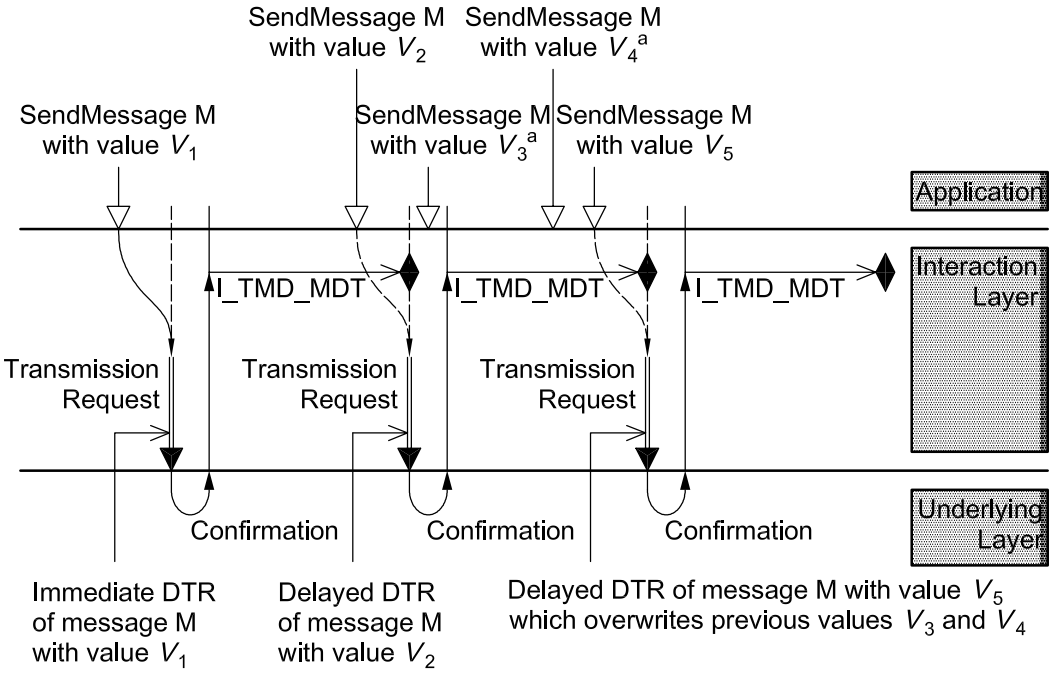


Figure 4 — Symbols used in figures

A minimum delay time between transmissions (I_TMD_MDT, greater than or equal to zero) shall be configured per I-PDU. If a transmission is requested before I_TMD_MDT expires, the next transmission is postponed until the delay time expires.

The minimum delay time for the next transmission starts the moment the previous transmission is confirmed. If a transmission is seen as erroneous because of Transmission Deadline Monitoring, the next transmission can start immediately.



DTR = Direct Transmission Request

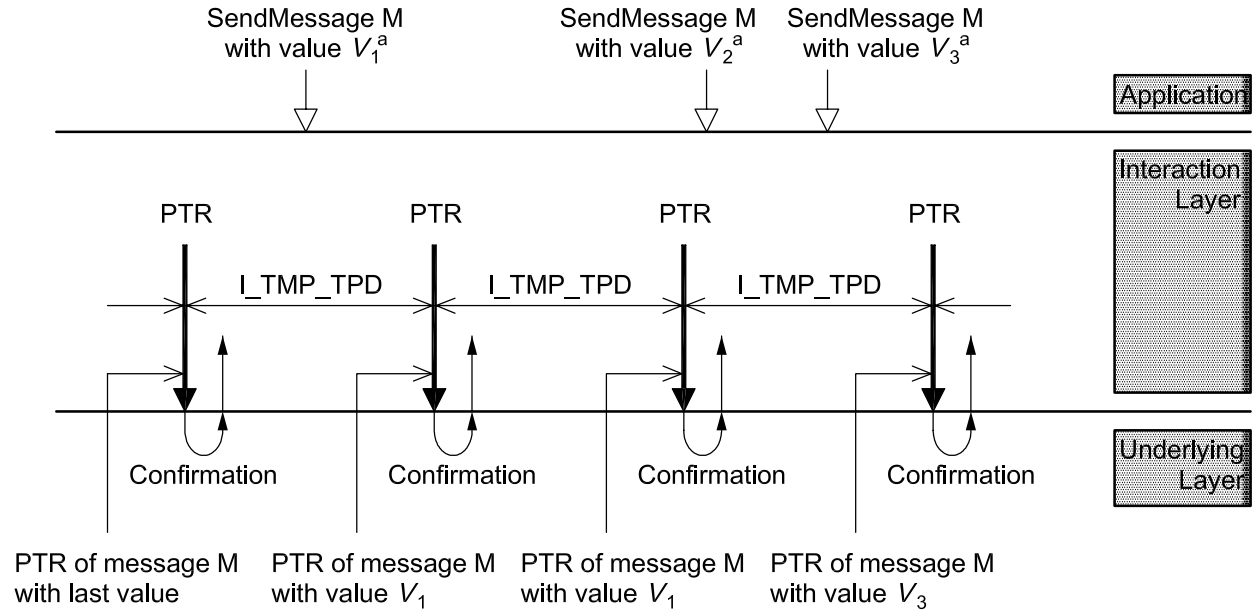
Figure 5 — Direct Transmission Mode with minimum delay time

3.3.4.3 Periodic Transmission Mode

In *Periodic Transmission Mode* the IL issues periodic transmission requests for an I-PDU to the underlying layer.

Each call to the API service *SendMessage* or *SendDynamicMessage* updates the message object with the message to be transmitted, but does not issue any transmission request to the underlying layer. The Periodic Transmission Mode ignores the transfer property of all messages contained in the I-PDU. (See Annex B for more information.)

The transmission is performed by repeatedly calling the appropriate service in the underlying layer with a period equal to the Periodic Transmission Mode Time Period (I_TMP_TPD).



PTR = Periodic Transmission Request

Figure 6 — Periodic Transmission Mode

3.3.4.4 Mixed Transmission Mode

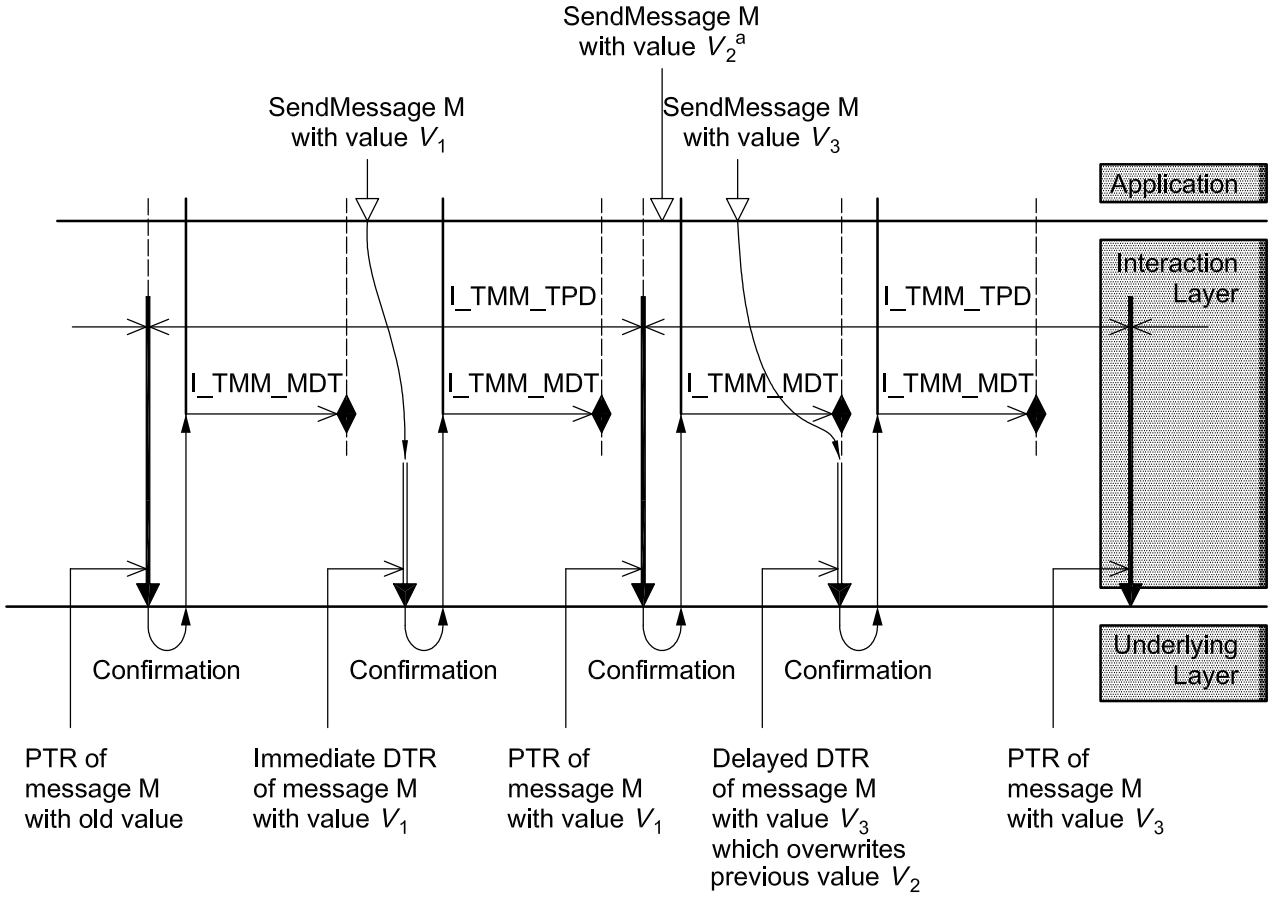
Mixed Transmission Mode is a combination of the Direct and the Periodic Transmission Modes.

The transmission is performed by repeatedly calling the appropriate service in the underlying layer with a period equal to the Mixed Transmission Mode Time Period (I_TMM_TPD).

Intermediate transmission of an I-PDU is caused by the transfer of any message assigned to this I-PDU with Triggered Transfer Property. The transfer is immediately followed by a transmission request from the IL to the underlying layer. These intermediate transmissions do not modify the base cycle (i.e. I_TMM_TPD).

A minimum delay time between transmissions (I_TMM_MDT, greater than or equal to zero) shall be configured. If transmissions are requested before I_TMM_MDT expires, the next transmission is postponed until the delay time expires.

The minimum delay time for the next transmission starts the moment the previous transmission is confirmed. If a transmission is seen as erroneous because of Transmission Deadline Monitoring, the next transmission can start immediately.



PTR = Periodic Transmission Request
DTR = Direct Transmission Request

Figure 7 — Mixed Transmission Mode with minimum delay time (simple cases)

An intermediate transmission request less than I_TMM_MDT before the next periodic transmission request (PTR) delays this PTR and possibly also subsequent PTRs, as shown in Figure 8.

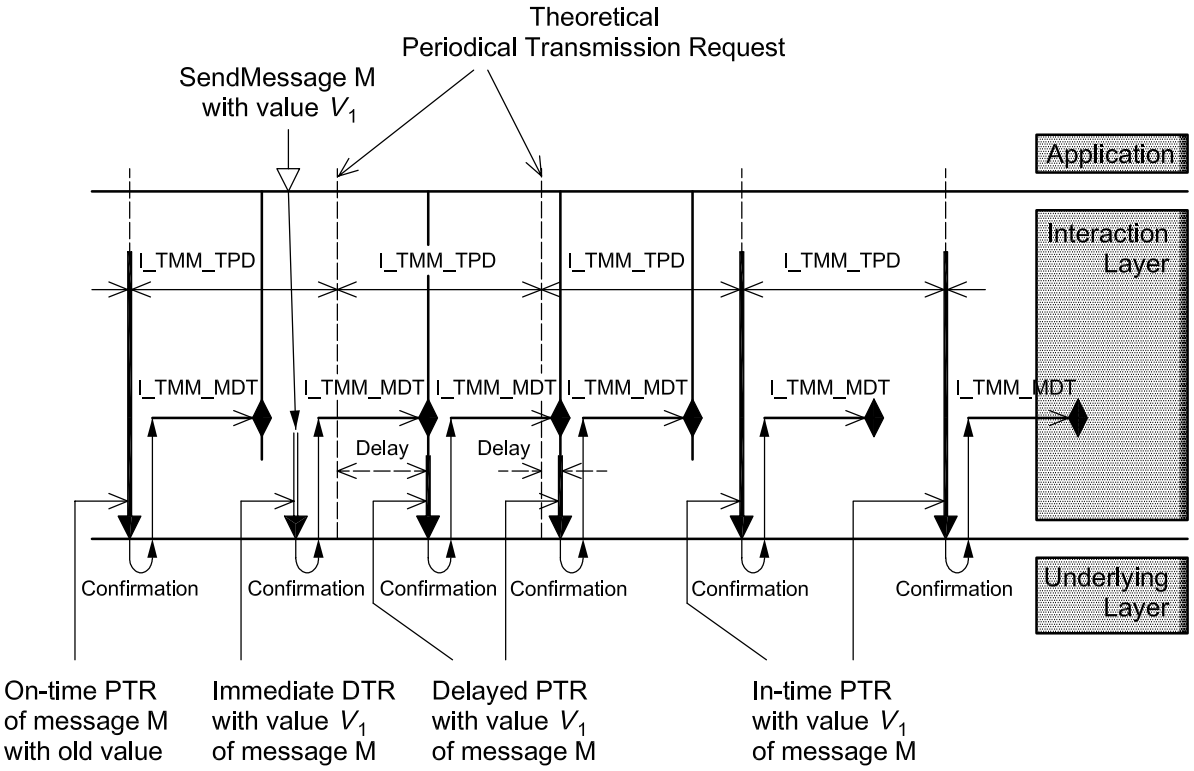


Figure 8 — Mixed Transmission Mode with minimum delay time (MDT delays PTR)

3.3.5 Activation/Deactivation of periodic transmission mechanism

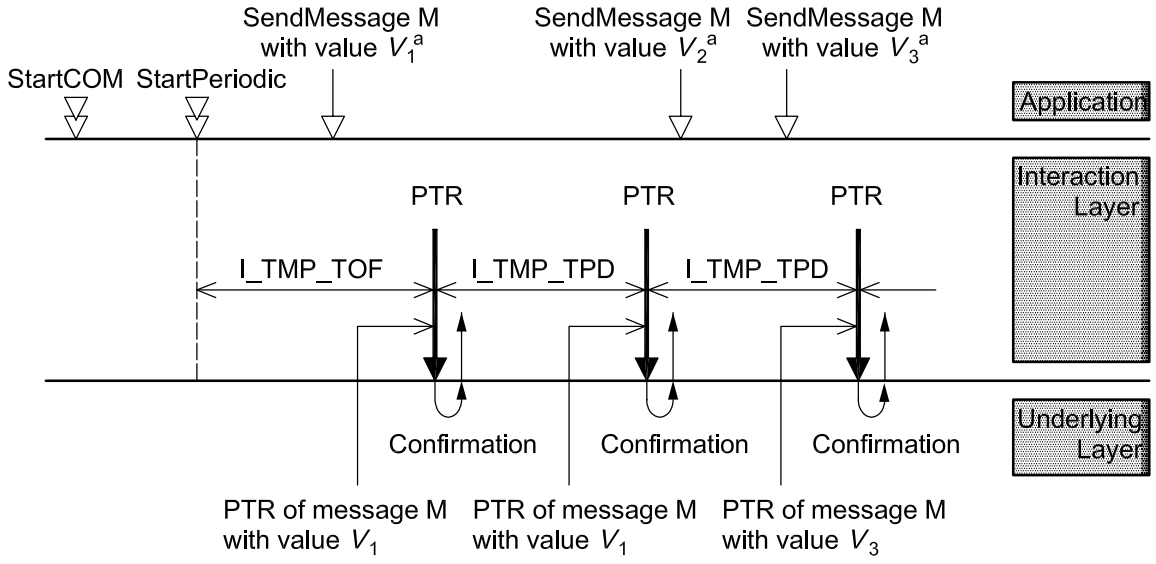
The periodic transmission mechanism in the Periodic and the Mixed Transmission Modes is activated by a call to the *StartPeriodic* API service. The *StartPeriodic* service initializes and starts the Periodic or the Mixed Transmission Mode Time Offset (I_TMP_TOF or I_TMM_TOF) timer.

The first transmission request is issued upon expiry of the Periodic or the Mixed Transmission Mode Time Offset (I_TMP_TOF or I_TMM_TOF).

StartPeriodic shall be called after the *StartCOM* API service has completed and once the message object is correctly initialized. The API service *InitMessage* can be used to perform this initialization (not shown in Figure 9).

The periodic transmission mechanism is stopped by means of the *StopPeriodic* API service.

The Periodic or the Mixed Transmission Mode time offset is configured per I-PDU.



PTR = Periodic Transmission Request

Figure 9 — Activation of the periodic transmission mechanism (example for an I-PDU with Periodic Transmission Mode)

3.3.6 Message filtering algorithm

Message filtering is used to suppress the transfer of messages. The IL compares the new message value to the last sent message value and only transfers the message if the filtering condition is met. All other message values are discarded.

For message filtering, the algorithms listed in Table 1 are supported.

No message filtering is performed for zero-length messages and dynamic-length messages.

3.4 Byte order conversion and message interpretation

The IL is responsible for the byte order conversion between the local CPU and the underlying layers and vice versa. Byte order conversion (big-endian to little-endian and vice versa) takes place on the sender side before messages are stored in the I-PDU and on the receiver side when they are retrieved from the I-PDU. Messages are configured either to remain untouched, or to be interpreted as C unsigned integer types and converted. No byte order conversion takes place on internal messages and dynamic-length messages.

The IL does not prescribe the byte order used in I-PDUs; different messages in the same I-PDU may have different byte orders.

On the sender side, for messages which are interpreted as integers, the most significant bits are truncated, if necessary.

On the receiver side, for messages which are interpreted as integers, the most significant bits are filled with 0, if necessary.

Dynamic-length messages are always interpreted as byte arrays.

3.4.1 Bit and byte numbering in I-PDUs and messages

An I-PDU is a sequence of bytes numbered from 0 upwards. Within an I-PDU byte, bits are numbered from 0 upwards with bit 0 being the least significant bit.

A message, at the moment it is packed to the I-PDU, is regarded as a sequence of bits numbered from 0 upwards with bit 0 being the least significant bit.

I-PDU bits are numbered counting from 0 upwards from bit 0 of byte 0 of the sequence of bytes constituting the I-PDU.

3.4.2 Little-endian byte order

A message placed at bit *n* of an I-PDU occupies bits *n*, *n+1*, *n+2*, etc. of the I-PDU up to the length of the message.

The least significant bit of the message (LSB, message bit *0*) is stored in I-PDU bit *n*.

The most significant bit of the message (MSB, message bit *i*) is stored in I-PDU bit *n+i*.

This byte order is called *little-endian byte order* (see Figure 10).

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15 ← 2	14 ← 1	13 LSB ← 0	12	11	10	9	8
Byte 2	23 ← 10	22 ← 9	21 ← 8	20 ← 7	19 ← 6	18 ← 5	17 ← 4	16 ← 3
Byte 3	31	30	29	28	27	26	25	24 MSB ← 11
Byte 4	39	38	37	36	35	34	33	32

Figure 10 — Little-endian byte order

3.4.3 Big-endian byte order

A message placed at bit *n* of an I-PDU occupies bits *n*, *n+1*, *n+2*, etc. of the I-PDU up to the length of the message or up to the next I-PDU byte boundary, whichever comes first.

If a message exceeds the boundary of I-PDU byte *m*, packing of message bits is continued from the least significant bit of I-PDU byte *m-1* upwards.

This byte order is called *big-endian byte order* (see Figure 11).

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15	14	13 MSB ← 11	← 10	← 9	← 8	← 7	← 6
Byte 2	23 ← 5	← 4	← 3	← 2	← 1	18 LSB ← 0	17	16
Byte 3	31	30	29	28	27	26	25	24
Byte 4	39	38	37	36	35	34	33	32

Figure 11 — Big-endian byte order

3.5 Deadline monitoring

3.5.1 Reception Deadline Monitoring

Reception Deadline Monitoring can be used to verify on the receiver side that periodic messages are received within the allowed time frame. This mechanism is configured per message and is performed by monitoring the reception of the I-PDU that contains the message.

Reception Deadline Monitoring is restricted to external communication.

The deadline monitoring mechanism monitors that a periodic message is received within a given time interval (I_DM_RX_TO).

The monitoring timer is cancelled and restarted by the IL upon each new reception from the underlying layer of the PDU that contains the message.

If there is no indication of the PDU's reception by the underlying layer, the time-out occurs and the timer is immediately restarted.

The timer for the first monitored time interval is started once message object initialization tasks are performed, i.e. after the *StartCOM* API has completed. Depending on system design constraints, a specific value (I_DM_FRX_TO) can be chosen for the first time-out interval.

The use of this mechanism is not restricted to monitoring the reception of messages (I-PDUs) transmitted using Periodic Transmission Mode, but also can be applied to messages (I-PDUs) sent using the Direct and Mixed Transmission Modes. Indirect NM or the application can be notified upon the expiry of a time-out.

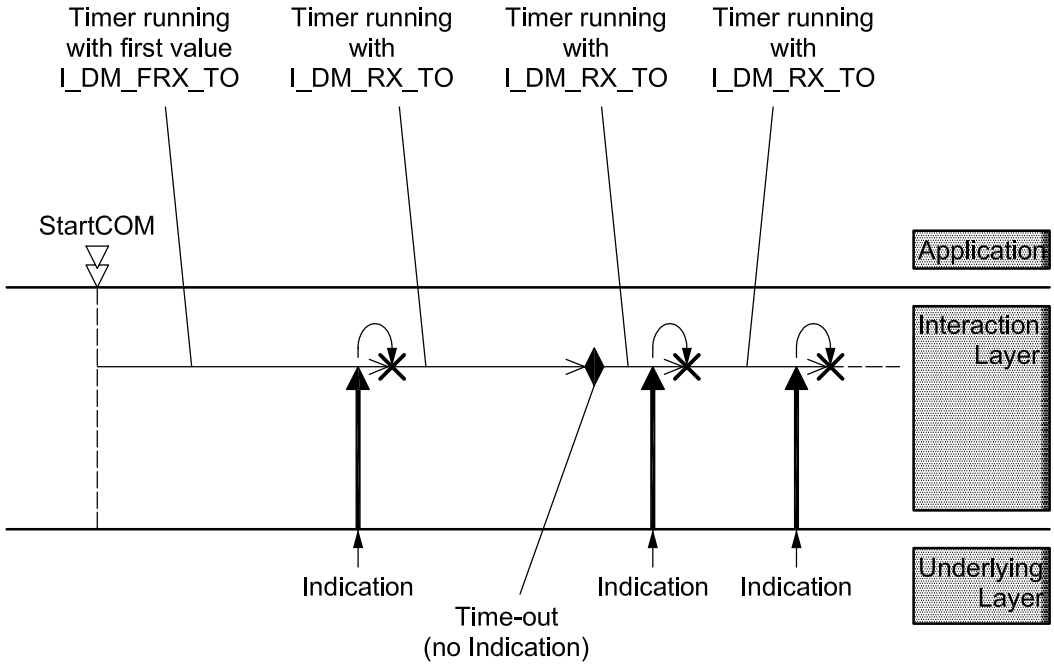


Figure 12 — Deadline Monitoring for periodic reception

3.5.2 Transmission Deadline Monitoring

3.5.2.1 General

This section defines mechanisms for monitoring the transmission of messages.

Deadline Monitoring on the sender side can be used to verify that transmission requests (periodic or otherwise) are followed by transmissions on the network within a given time frame.

Whether Transmission Deadline Monitoring shall be performed can be configured separately for each message. However, the IL performs Transmission Deadline Monitoring per I-PDU. Therefore, the time-out period is a property of the I-PDU.

For messages using Triggered Transfer Property, transmission monitoring is available for any transmission mode. For messages using Pending Transfer Property, transmission monitoring is available for Periodic Transmission Mode and the periodic part of Mixed Transmission Mode.

3.5.2.2 Direct Transmission Mode

The deadline monitoring mechanism monitors each call to *SendMessage*, *SendDynamicMessage* or *SendZeroMessage* and checks that a confirmation by the underlying layer occurs within a given time interval (I_DM_TMD_TO).

The monitoring timer is started upon completion of the call to the *SendMessage*, *SendDynamicMessage* or *SendZeroMessage* API service.

The timer is cancelled upon confirmation of the transmission by the underlying layer and the application is notified using the appropriate notification mechanism.

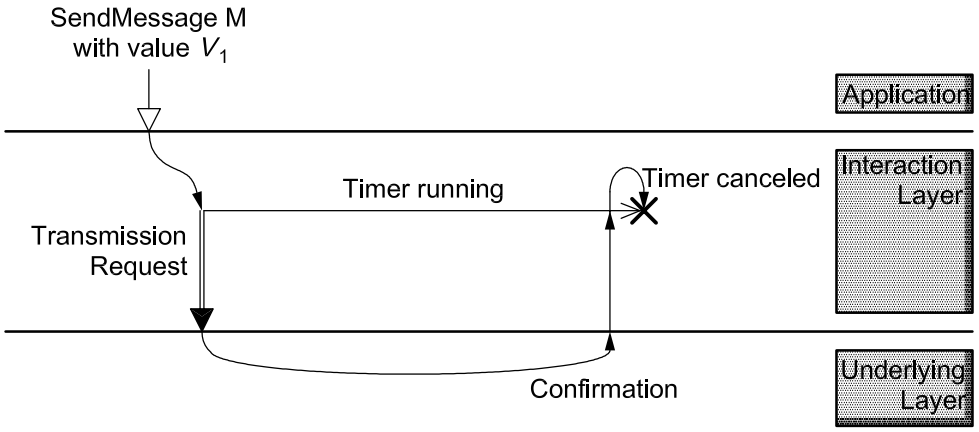


Figure 13 — Direct Transmission Mode: example of a successful transmission

If the transmission does not occur, i.e. if there is no confirmation of the I-PDU's transmission by the underlying layer, the time-out occurs and the application is notified using the appropriate notification mechanism.

The IL does not retry transmission requests upon the occurrence of a time-out. It is up to the application to decide upon the appropriate actions to be taken.

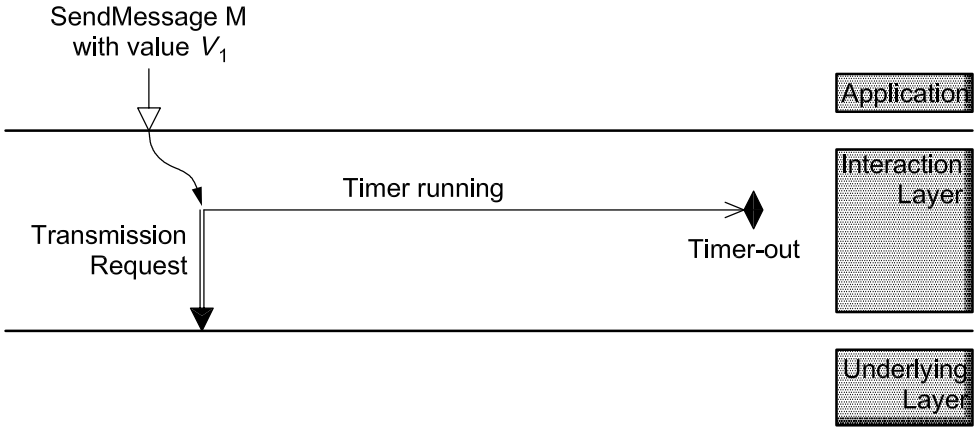


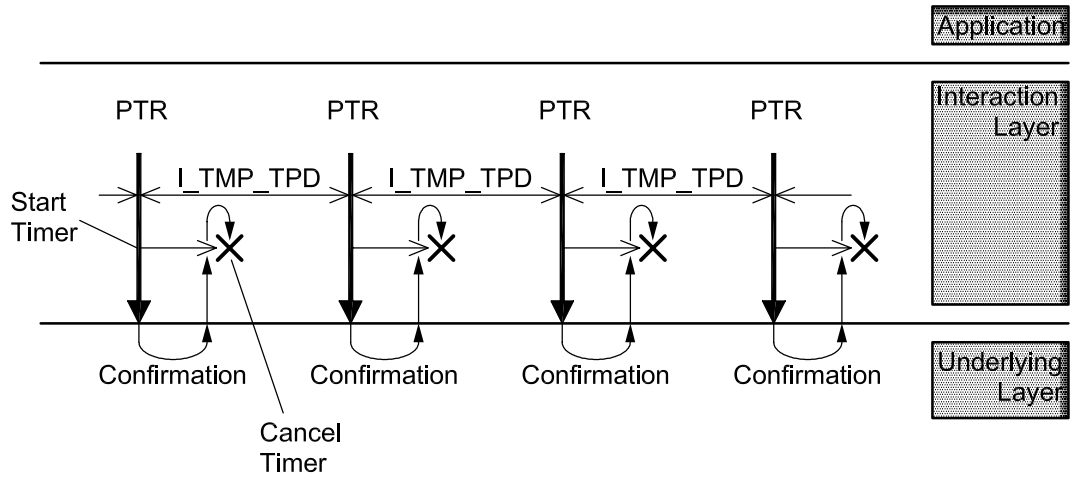
Figure 14 — Direct Transmission Mode: example of a failed transmission

3.5.2.3 Periodic Transmission Mode

The deadline monitoring mechanism monitors that an I-PDU is transmitted within a given time interval. The period of the time-out interval (I_DM_TMP_TO) can be greater than the transmission period, depending on system design constraints.

The monitoring timer is started after each periodic transmission request if it is not currently running (i.e. if it is the first time the timer has been started, or if the timer was previously cancelled).

The timer for the corresponding monitored time interval (I_DM_TMP_TO) is cancelled by the confirmation of any transmission of the monitored I-PDU by the underlying layer.



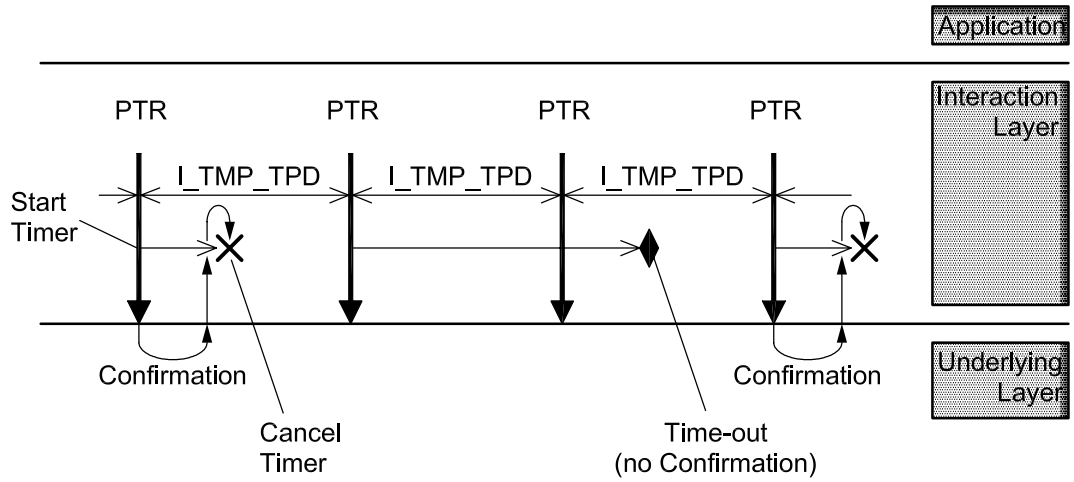
PTR = Periodic Transmission Request

Figure 15 — Periodic Transmission Mode: successful transmission

If the transmission does not occur, i.e. if there is no confirmation of the I-PDU's transmission by the underlying layer, the time-out occurs and the application is notified using the appropriate notification mechanism.

If the duration of the monitored time interval is equivalent to more than one transmission period, the timer is not restarted after each transmission request: the timer (I_DM_TMP_TO) is only restarted upon a transmission request if the previous timer has expired or has been cancelled.

The IL does not retry a transmission request upon the occurrence of a time-out. Transmission requests are still performed on the same cyclic basis.



PTR = Periodic Transmission Request

Figure 16 — Periodic Transmission Mode: failed transmission

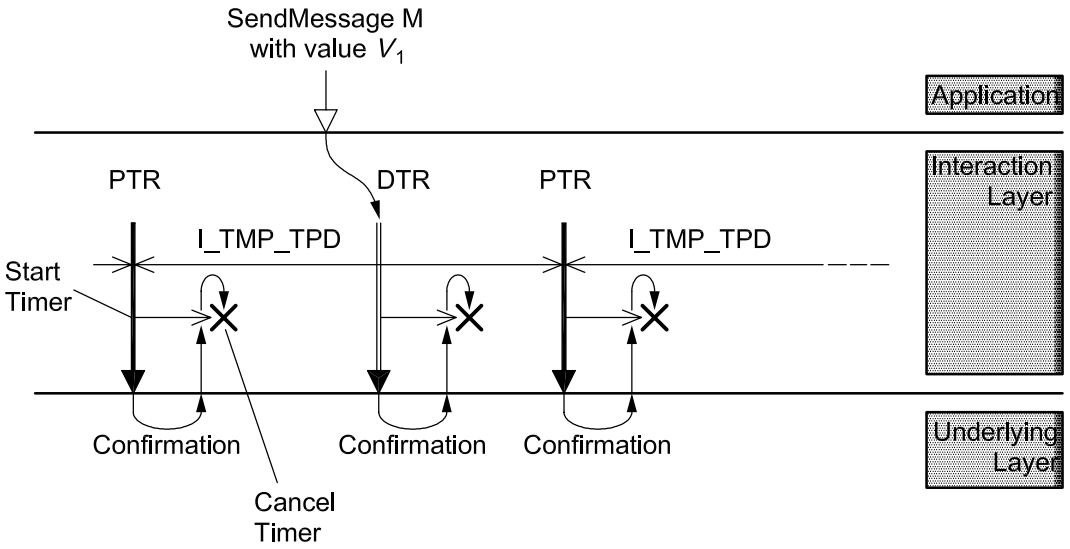
The application or Indirect NM can be notified upon the occurrence of the time-out or upon the successful transfer of the I-PDU within the allowed time interval.

3.5.2.4 Mixed Transmission Mode

The deadline monitoring mechanism monitors that an I-PDU is transmitted within a given monitored time interval (I_DM_TMM_TO).

The timer (I_DM_TMM_TO) is started after each transmission request if it is not currently running (i.e. if it is the first time the timer has been started, or if the timer was previously cancelled).

The timer is cancelled by the confirmation of any transmission of the monitored I-PDU by the underlying layer.



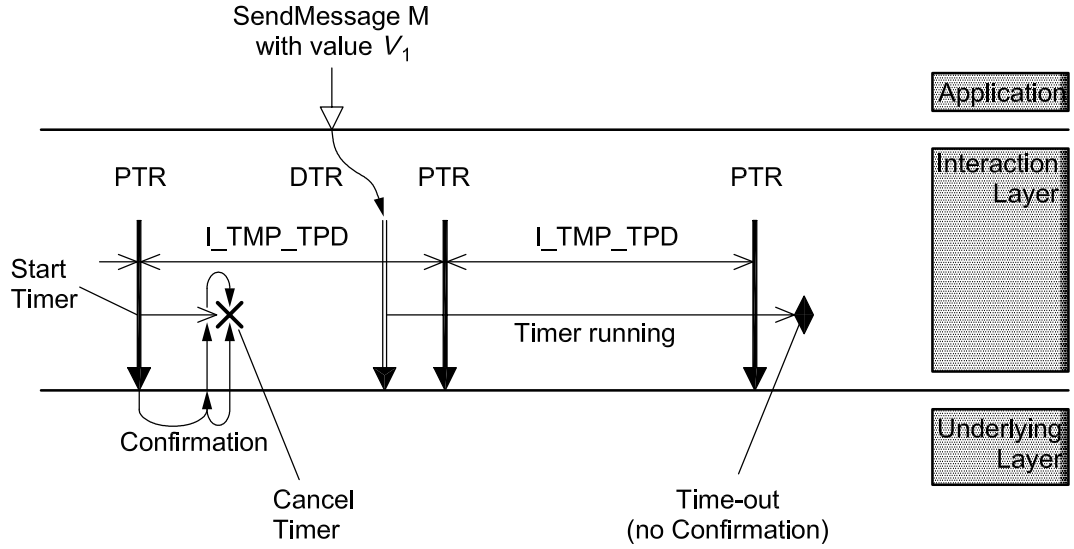
PTR = Periodic Transmission Request
DTR = Direct Transmission Request

Figure 17 — Mixed Transmission Mode: successful transmissions

If the duration of the monitored time interval is equivalent to more than one transmission period, the timer is not restarted after each transmission request: the timer is only restarted upon a transmission request if the previous timer has expired or has been cancelled.

The IL does not retry transmission requests upon the occurrence of a time-out. Transmission requests are still performed on the same cyclic basis or if a message with Triggered Transfer Property updates the I-PDU.

The application or Indirect NM can be notified upon the occurrence of the time-out or upon the successful transfer of the I-PDU within the allowed time interval.



PTR = Periodic Transmission Request
DTR = Direct Transmission Request

Figure 18 — Mixed Transmission Mode: failed transmissions

3.6 Notification

3.6.1 General

This subclause defines the notification mechanisms available to the application to determine the final status of a previously called send or receive operation.

The application is notified as soon as a specific event has occurred, e.g. the user does not need to call a specific ISO 17356-4 API service in advance to ensure that the notification scheme is active.

Notification is always a conditional notification. This means that, in the case of filtering, notification is only performed if the (transmitted or received) data is not discarded by the filtering mechanism. Likewise, notification on the receiver side is not performed if a queued message is discarded because of a buffer overflow condition.

Notification is configured per message object on both the sender and receiver sides. It is performed by monitoring the I-PDU that contains the message for transmission, and by monitoring the message object for reception.

3.6.2 Notification classes

This part of ISO 17356 supports four notification classes for message transmission and reception. Classes 1 and 3 notify the receiver of a message whereas Classes 2 and 4 notify the sender of a message.

All classes are supported for external communication.

For internal communication, only Class 1 is supported.

1) **Notification Class 1:** Message Reception — The configured notification mechanism is invoked immediately after the message has been stored in the receiving message object.

2) **Notification Class 2**: Message Transmission — The configured notification mechanism is invoked immediately after successful transmission of the I-PDU containing the message.

3) **Notification Class 3**: Message Reception Error — The configured notification mechanism is invoked immediately after a message reception error has been detected either by the deadline monitoring mechanism or via an error code provided by the indication service of the underlying layer.

4) **Notification Class 4**: Message Transmission Error — The configured notification mechanism is invoked immediately after a message transmission error has been detected either by the deadline monitoring mechanism or via an error code provided by the confirmation service of the underlying layer.

3.6.3 Notification mechanisms

The following notification mechanisms are provided [1)]:

1) Callback routine — The IL calls a callback routine provided by the application.

2) Flag — The IL sets a flag that can be checked by the application by means of the *ReadFlag* API service (*ReadFlag* returns COM_TRUE if the flag is set, otherwise it returns COM_FALSE). Resetting the flag is performed by the application by means of the *ResetFlag* API service. Additionally, calls to *ReceiveMessage* and *ReceiveDynamicMessage* reset flags defined for Notification Classes 1 and 3 and calls to *SendMessage*, *SendDynamicMessage* and *SendZeroMessage* reset flags defined for Notification Classes 2 and 4.

3) Task — The IL activates an application task.

4) Event — The IL sets an event for an application task.

Only one type of notification mechanism can be defined for a given sender or receiver message object and a given notification class. All notification mechanisms are available for all notification classes.

Except for *StartCOM* and *StopCOM*, the use of all ISO 17356-4 API functions is allowed in callback routines. The user shall take care of problems which can arise because of nesting of callbacks (stack size etc.).

3.6.4 Interface for callback routines

Within the application, a callback routine is defined according to the following template:

```
COMCallback(CallbackRoutineName)
{
}
```

No parameters are passed to a callback routine and they do not have a return value.

A callback routine runs either on interrupt level or on task level. Thus, the restrictions in ISO 17356-3 concerning usage of system functions for interrupt service routines as well as for tasks apply.

3.7 Communication system management

3.7.1 Initialization/Shutdown

The start-up of a distributed system depends heavily on the communication protocol used and can only be specified with detailed knowledge of this protocol. Therefore, the description of the communication protocol specific API is not defined within this part of ISO 17356. It is assumed that all underlying layers are correctly started and the necessary communication protocols are running.

---

1) An additional notification mechanism is supported for indirect NM (see 3.9.2).

This part of ISO 17356 provides the following services to start up and shut down communication:

— StartCOM: This service initializes internal ISO 17356-4 data areas, calls message initialization routines and starts the ISO 17356-4 module.

— StopCOM: This service is used to terminate a session of COM and release resources where applicable.

— StartPeriodic and StopPeriodic: These services start or stop the periodic transmission of all messages using the Periodic or the Mixed Transmission Mode. It is sometimes useful to suspend periodic activity without necessarily closing down the whole of COM.

NOTE 1 *StartCOM* does not automatically enable periodic transmission.

NOTE 2 *StopCOM* terminates periodic transmission.

— InitMessage: This service allows the application to initialize messages with arbitrary values.

Once the kernel has started, an application calls *StartCOM*. This service is intended to allocate and initialize system resources used by the ISO 17356-4 module. If configured in ISO 17356-6, *StartCOM* calls a user-supplied function *StartCOMExtension*.

For queued messages *StartCOM* initializes the number of received messages to 0.

Unqueued messages can be initialized in three ways: no initial value specified in the ISO 17356-6 file, initial value specified in the ISO 17356-6 file and explicitly via the *InitMessage* call.

If a message has no initial value specified in the ISO 17356-6 file then StartCOM initializes it to the value 0.

If a message has an initial value specified in the ISO 17356-6 file, then the message is initialized to that value. However, note that ISO 17356-6 only allows the specification of a limited range of unsigned integer initialization values. This means that ISO 17356-6 can only be used to initialize messages that correspond to unsigned integer types within ISO 17356-6's range of values.

Messages defined to be initialized with no initial value, or with values specified in the ISO 17356-6 file, shall be initialized by *StartCOM* before *StartCOM* calls *StartCOMExtension.*

— *InitMessage* can be used to initialize any message with any legal value. Therefore, InitMessage can also be used to initialize messages that are too large or complex for their initial value to be specified in ISO 17356-6.

— *InitMessage* can be called at any point in the application's execution after *StartCOM* has been called and before *StopCOM* is called but is typically used in *StartCOMExtension*.

— *InitMessage* can be used to re-initialize any message after it has been initialized to 0 or a value specified in the ISO 17356-6 file.

For all three ways of initializing a message, the following operations take place:

— For external transmit messages, the message field in the I-PDU and old_value are set to the value specified.

— For internal transmit messages, no initialization takes place.

— For receive messages, the message object for an unqueued message is set to the value specified. If a filter algorithm using old_value (see Table 1) is specified for either unqueued or queued messages, old_value is set to the value specified.

In the case of dynamic-length messages, the *InitMessage* call initializes the entire message and the length field is initialized to the message's maximum length.

For queued messages, *InitMessage* sets the number of received messages to 0.

StartCOM supports the possibility of starting communication in different configurations. To do this, a parameter is transferred in the call to *StartCOM*.

StartPeriodic and *StopPeriodic* shall be used to control the periodic transmission of I-PDUs with the Periodic or the Mixed Transmission Mode.

StopCOM is designed in such a way that an application can terminate communication in order to release its resources. This part of ISO 17356 can be restarted with the *StartCOM* service afterwards, thus the data are reset to the initial values. *StopCOM* does not prevent message corruption; unread messages are inaccessible to the application and are therefore lost.

Before *StartCOM* is called for the first time, and after *StopCOM* has been successfully completed, the behaviour of all ISO 17356-4 calls other than *StartCOM* is undefined by this part of ISO 17356. However, the vendor shall define the behaviour of all ISO 17356-4 calls under these circumstances.

3.7.2 Error handling

3.7.2.1 General remarks

An error service is provided to handle temporarily and permanently occurring errors within ISO 17356-4. Its basic framework is predefined and shall be completed by the user. This gives the user a choice of efficient centralized or decentralized error handling.

Two different kinds of errors are distinguished:

— Application errors: The IL could not execute the requested service correctly, but assumes the correctness of its internal data. In this case, centralized error treatment is called. Additionally the IL returns the error by the status information for decentralized error treatment. It is up to the user to decide what to do depending on which error has occurred.

— Fatal errors: The IL can no longer assume correctness of its internal data. In this case, the IL calls the centralized system shutdown.

All these error services are invoked with a parameter that specifies the error.

This part of ISO 17356 offers two levels of error checking:

— Extended error checking: Extended error checking is provided to support the testing of incompletely debugged applications during the development phase. It allows enhanced plausibility checks, but requires more execution time and more memory space than standard error checking. The range of status codes returned by ISO 17356-4 API services on Extended error checking level is called Extended Status.

— Standard error checking: Standard error checking is used in a fully debugged application system during the production phase. The range of status codes returned by ISO 17356-4 API services on Standard error checking level is called Standard Status.

The return values of the API services have precedence over the output parameters. If an API service returns an error, the values of the output parameters are undefined.

3.7.2.2 Error hook routine

The ISO 17356-4 error hook routine (*COMErrorHook*) is called if an ISO 17356-4 service rerturns a *StatusType* value not equal to E_OK. The hook routine *COMErrorHook* is not called if an ISO 17356-4 service is called from the *COMErrorHook* itself (i.e. a recursive call to the ISO 17356-4 error hook never occurs). Any errors caused by a ISO 17356-4 service called from within *COMErrorHook* can only be detected by evaluating the service's return value.

This hook routine is:

— called by the IL, in a context depending on the implementation;

— not interruptible by category 2 interrupt service routines (see ISO 17356-3);

— part of the IL;

— implemented by the user with user-defined functionality;

— standardized in interface, but not standardized in functionality and therefore usually not portable;

— only allowed to use the API functions GetMessageStatus and COMErrorGetServiceId and the parameter access macros COMError_Name1_Name2; and

— mandatory, but configurable via ISO 17356-6.

3.7.2.3 Error management

To allow for effective error management in *COMErrorHook*, the user can access additional information.

The macro *COMErrorGetServiceId* provides an identifier indicating the service that gave rise to the error. The service identifier is of type *COMServiceIdType*. Possible values are COMServiceId_xxxx, where xxxx is the name of the service. Implementation of *COMErrorGetServiceId* is mandatory. If the service that caused *COMErrorHook* to be called has parameters, then these can be accessed using the following access macro name building scheme. The macro names consist of a fixed prefix and two components *COMError_Name1_Name2* where:

— COMError: is the fixed prefix;

— Name1: is the name of the service; and

— Name2: is the name of the parameter.

For example, the macros to access the parameters of *SendMessage* are:

— COMError_SendMessage_Message(); and

— COMError_SendMessage_DataRef().

The macro to access the first parameter of a service is mandatory if the parameter is the message identifier of a message. For optimization purposes, the macro access can be switched off within ISO 17356-6.

3.8 Functional model of the Interaction Layer

The following figures illustrate the behaviour of the IL for external reception, external transmission and internal communication. They provide the context for the concepts introduced in the preceding sections. Notification mechanisms are hinted at, but not shown in full detail. These models are presented for better understanding, but do not imply in any way a particular implementation of the IL. Depending on system constraints, an optimized model could be implemented.

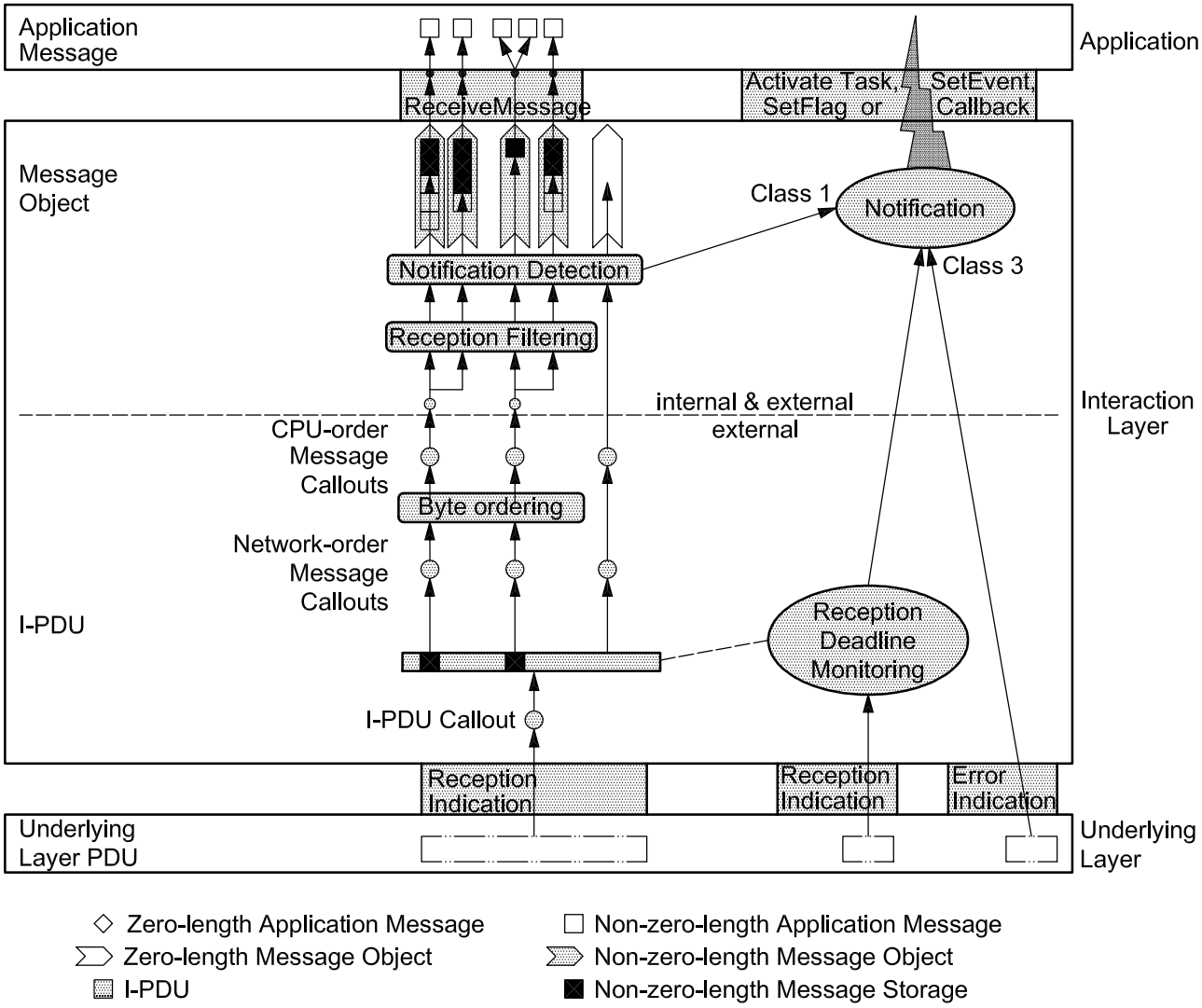


Figure 19 — IL model for external reception

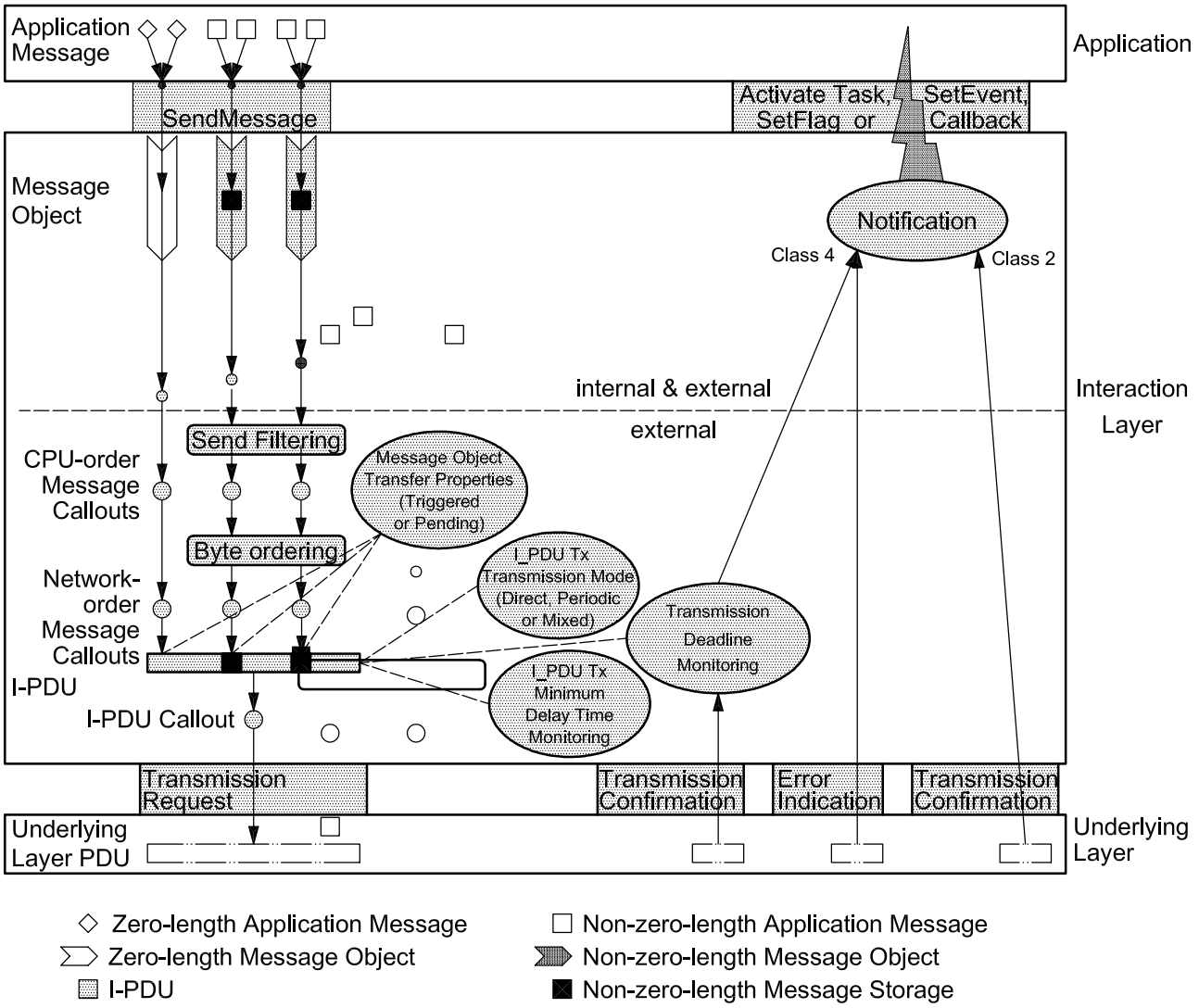


Figure 20 — IL model for external transmission

Figure 21 shows a model for internal communication (sender and receiver using the same IL), as well as for external transmission.

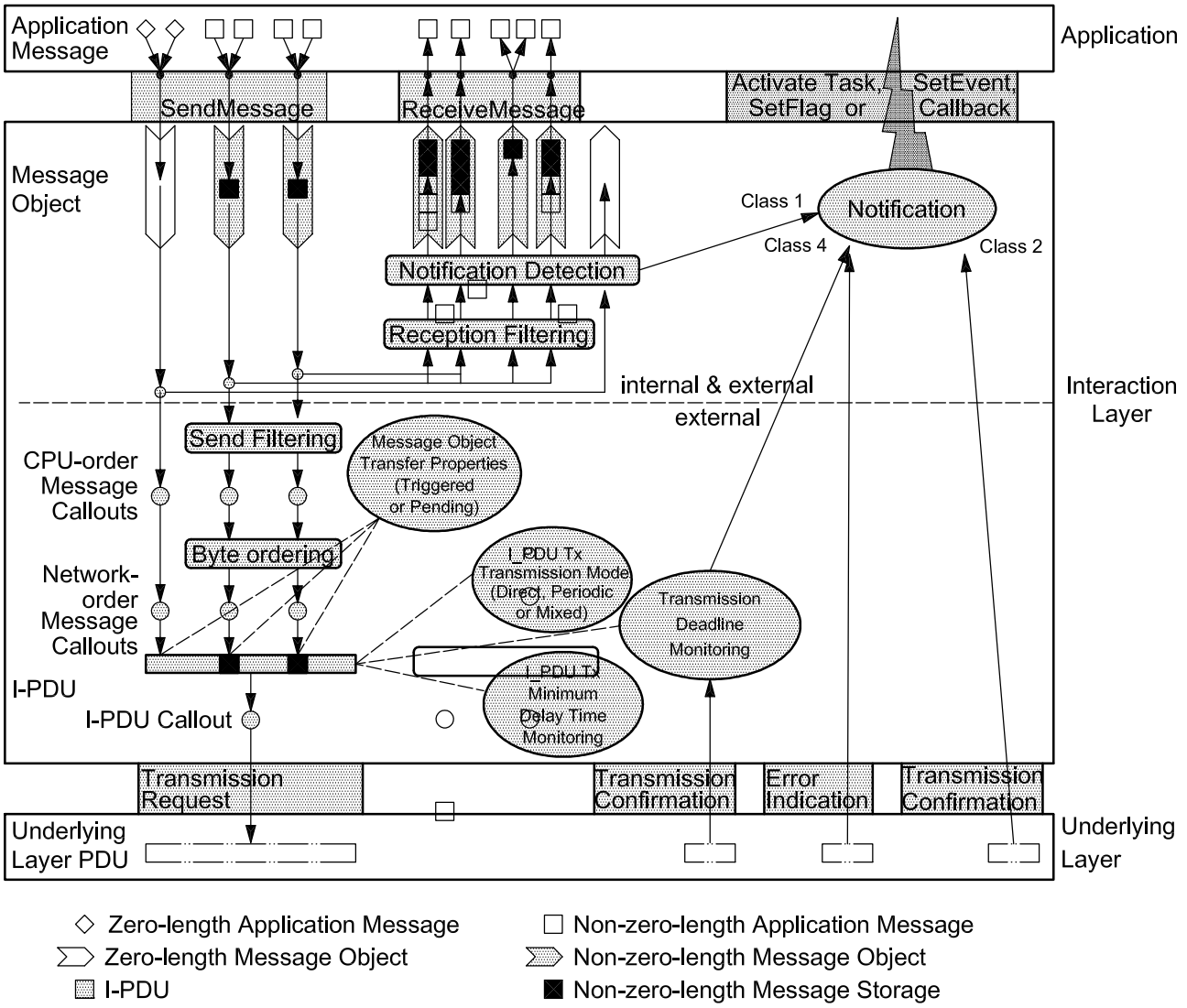


Figure 21 — IL model for internal communication and external transmission

3.9 Interfaces

3.9.1 General

The system service interface is ISO/ANSI-C. Its implementation is normally a function call, but may also be solved differently, by using C pre-processor macros, for example. A specific type of implementation cannot be assumed.

ISO 17356-4 services may internally call ISO 17356-3 services. When this part of ISO 17356 uses ISO 17356-3 services internally, any additional restrictions imposed upon the application by ISO 17356-3 are also imposed upon this part of ISO 17356. The return value of the ISO 17356 API services has precedence over the output parameters.

If an ISO 17356 API service returns an error, the values of the output parameters are undefined. The sequence of error checking within this part of ISO 17356 is not specified. Whenever multiple errors occur, it is implementation-dependent which status is returned to the application.

3.9.2 Interface to Indirect NM

3.9.2.1 Basics

The following services are provided by Indirect NM as callback functions for this part of ISO 17356 to inform Indirect NM of deadline monitoring results. They provide a fifth notification mechanism, *NMCallback*. This notification mechanism is identical to the *COMCallback* mechanism described in 3.6.3 except that the interface complies to the definition of *I_MessageTransfer.ind* and *I_MessageTimeOut.ind*, that is:

— NMCallback routines have no return value; and

— NMCallback routines pass a 16-bit unsigned integer value as parameter.

Both the name of the NMCallback routine and the value of the parameter passed to it are statically defined in ISO 17356-6.

To allow for proper configuration, implementations of Indirect NM shall describe implementation-specific naming conventions (what are the C language names for *I_MessageTransfer.ind* and *I_MessageTimeOut.ind*) and parameter conventions (how do parameter values map to monitored I-PDUs).

3.9.2.2 I-PDU transfer indication

Service name: **`I_MessageTransfer`**

Service primitive: I_MessageTransfer.ind (<MonitoredIPDU>)

Parameter (in):

MonitoredIPDU 16-bit unsigned integer value identifying the I-PDU to be monitored

Parameter (out): None

Description: This part of ISO 17356 informs Indirect NM via the service primitive *I_MessageTransfer.ind* that a monitored I-PDU has been received from a remote node or that a monitored I-PDU has been transmitted by the local node.

3.9.2.3 I-PDU time-out indication

Service name: **`I_MessageTimeOut`**

Service primitive: I_MessageTimeOut.ind (<MonitoredIPDU>)

Parameter (in):

MonitoredIPDU 16-bit unsigned integer value identifying the I-PDU to be monitored

Parameter (out): None

Description: This part of ISO 17356 informs Indirect NM via the service primitive *I_MessageTimeOut.ind* that a time-out has occurred for a monitored I-PDU received from a remote node or for a monitored I-PDU transmitted by the local node.

3.9.3 Application Program Interface (API)

3.9.3.1 Service parameter types

This subclause describes the types of API service in/out parameters.

3.9.3.1.1 StatusType

Description:

This part of ISO 17356 defines communication-specific status codes. The following naming conventions shall apply:

The names of all status codes which are applicable throughout the whole of ISO 17356 (universal status codes) shall start with E_. There is only one universal status code: E_OK.

The names of all status codes which are defined by this part of ISO 17356 (communication-specific status codes) shall start with E_COM_, e.g. E_COM_NOMSG.

The following table lists the universal status codes used by this part of ISO 17356 and the communication-specific status codes defined by this part of ISO 17356:

Table 2 — Status codes used and/or defined by this part of ISO 17356

Status code	Description
E_OK	Service call has succeeded.
E_COM_ID	Given message or mode identifier is out of range or invalid.
E_COM_LENGTH	Given data length is out of range.
E_COM_LIMIT	Overflow of message queue.
E_COM_NOMSG	Message queue is empty.

The system designer can add implementation-specific status codes for this part of ISO 17356. The names of all implementation-specific status codes shall start with E_COM_SYS_, e.g. E_COM_SYS_DISCONNECTED.

An implementation-specific status code may either yield an error which is encountered by the ISO 17356-4 service when calling an ISO 17356 service such as, e.g. *ActivateTask*, or a specific error of the ISO 17356-4 service itself. In the former case, it is recommended that the implementation-specific status code returned is that of the respective ISO 17356 service. Otherwise, the implementation-specific status code shall be a status code in the system-reserved number space of this part of ISO 17356 (see ISO 17356-2).

All implementation-specific status codes shall be described in the vendor-specific documentation of an implementation.

Refer to ISO 17356-2 for more information on the *StatusType* parameter.

3.9.3.1.2 MessageIdentifier

Type: Scalar

Range: Application-specific, depends on the range of message identifiers

Description: ISO 17356-4 message object identifier

3.9.3.1.3 ApplicationDataRef

Type: Reference to a data field in the application

Range: Implementation-specific

Description: Pointer to the data field of an application message

3.9.3.1.4 COMLengthType

Type: Scalar

Range: Depends on the communication protocol used

Description: Data length

3.9.3.1.5 LengthRef

Type: Reference to scalar

Range: Depends on the communication protocol used

Description: Pointer to a data field containing length information

3.9.3.1.6 FlagValue

Type: Enumeration

Range: COM_FALSE, COM_TRUE

Description: Current state of a message flag

3.9.3.1.7 COMApplicationModeType

Type: Scalar

Range: Application-specific, depends on the number of ISO 17356-4 application modes

Description: Identifier for selected ISO 17356-4 application mode

3.9.3.1.8 COMShutdownModeType

Type: Scalar

Range: COM_SHUTDOWN_IMMEDIATE

Description: Identifier for selected ISO 17356-4 shutdown mode

3.9.3.1.9 CalloutReturnType

Type: Enumeration

Range: COM_FALSE, COM_TRUE

Description: Indicates at the exit of a callout whether the IL shall continue or abandon further processing of the current message or I-PDU

3.9.3.1.10 COMServiceIdType

Type: Enumeration

Range: COMServiceId_xx with xx being the name of an ISO 17356-4 service

Description: Unique identifier of an ISO 17356-4 service. Example: *COMServiceId_SendMessage*

3.9.3.2 Start-up services

3.9.3.2.1 StartCOM

Service name: `StartCOM`

Syntax: StatusType StartCOM (COMApplicationModeType <Mode>)

Parameter (in):

Mode: ISO 17356-4 application mode

Parameter (out): None

Description: The service *StartCOM* starts and initializes the ISO 17356-4 implementation in the requested application mode.

If *StartCOM* fails, initialization of the ISO 17356-4 implementation aborts and *StartCOM* returns a status code as specified below.

StartCOM shall be called from within a task if an operating system that conforms to ISO 17356-3 is used.

Before returning, the service *StartCOM* calls the application function *StartCOMExtension*.

Caveats: The hardware and low-level resources used by this part of ISO 17356 shall be initialized before *StartCOM* is called, otherwise undefined behaviour results.

StartCOM does not enable periodic transmission of messages. If needed, *StartPeriodic* can be called from *StartCOMExtension*. *StartCOM* does not stop periodic transmission when *StartCOMExtension* returns.

StartCOM returns the status code returned by *StartCOMExtension* if this is different from E_OK.

Status:

Standard:

— This service returns E_OK if the initialization completed successfully.

— This service returns an implementation-specific status code if the initialization was not completed successfully.

Extended:

In addition to the standard status codes defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Mode> is out of range.

3.9.3.2.2 StopCOM

Service name: `StopCOM`

Syntax: StatusType StopCOM (COMShutdownModeType <Mode>)

Parameter (in):

Mode: **COM_SHUTDOWN_IMMEDIATE**

The shutdown occurs immediately without waiting for pending operations to complete.

Parameter (out): None

Description: The service *StopCOM* causes all ISO 17356-4 activity to cease immediately. All resources used by this part of ISO 17356 are returned or left in an inactive state. Data loss is possible.

StopCOM stops all periodic transmission of messages.

When *StopCOM* completes successfully, the system is left in a state in which *StartCOM* can be called to re-initialize COM.

Status:

Standard:

— This service returns E_OK if COM was shut down successfully.

— This service returns an implementation-specific status code if the shutdown was not completed successfully.

Extended:

In addition to the standard status codes defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Mode> is out of range.

3.9.3.2.3 GetCOMApplicationMode

Service name: `GetCOMApplicationMode`

Syntax: COMApplicationModeType GetCOMApplicationMode (void)

Parameter (in): None

Parameter (out): None

Description: The service *GetCOMApplicationMode* returns the current ISO 17356-4 application mode. It may be used to write mode-dependent application code.

Particularities: If *GetCOMApplicationMode* is called before *StartCOM* is called, an implementation-specific code shall be returned (the ISO 17356-4 application mode is undefined).

Return value: Current ISO 17356-4 application mode.

3.9.3.2.4 InitMessage

Service name: **`InitMessage`**

Syntax:
```
StatusType InitMessage (
    MessageIdentifier <Message>,
    ApplicationDataRef <DataRef>
    )
```

Parameter (in):

Message: Message identifier (C identifier)

DataRef: Reference to the application's message initialization data

Parameter (out): none

Description: The service *InitMessage* initializes the message object identified by <Message> with the application data referenced by the <DataRef> parameter.

Particularities: This function may be called in *StartCOMExtension* in order to change the default initialization.

For dynamic-length messages, the length of the message is initialized to its maximum.

If *InitMessage* initializes a transmission message object directly in the I-PDU, additionally byte order conversion is performed and both the CPU-order and the Network-order Message Callouts are called.

Status:

Standard:

— This service returns E_OK if the initialization of the message object completed successfully.

— This service returns an implementation-specific status code if the initialization did not complete successfully.

Extended:

In addition to the standard status code defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or refers to a zero-length message or to an internal transmit message.

3.9.3.2.5 StartPeriodic

Service name: **`StartPeriodic`**

Syntax: StatusType StartPeriodic (void)

Parameter (in): None

Parameter (out): None

Description: The service *StartPeriodic* starts periodic transmission of all messages using either the Periodic or the Mixed Transmission Modes, unless periodic transmission is already started for these messages.

Particularities: Each call to *StartPeriodic* re-initializes and re-starts periodic transmission completely, i.e. taking into account defined time offsets.

Status:

Standard and Extended:

— This service returns E_OK if periodic transmission was started successfully.

— This service returns an implementation-specific status code if starting of periodic transmission was not completed successfully.

3.9.3.2.6 StopPeriodic

Service name: **`StopPeriodic`**

Syntax: StatusType StopPeriodic (void)

Parameter (in): None

Parameter (out): None

Description: The service *StopPeriodic* stops periodic transmission of all messages using either the Periodic or the Mixed Transmission Modes, unless periodic transmission is already stopped for these messages.

When *StopPeriodic* has completed successfully, the system is left in a state in which *StartPeriodic* can be called to restart periodic transmission of all messages using either the Periodic or the Mixed Transmission Modes.

Status:

Standard and Extended:

— This service returns E_OK if periodic transmission was stopped successfully.

— This service returns an implementation-specific status code if stopping periodic transmission was not completed successfully.

3.9.3.3 Notification mechanism support services

3.9.3.3.1 ReadFlag

Service name: **`ReadFlag`**

Syntax: FlagValue ReadFlag_<Flag>()

Parameter (in): None

Parameter (out): None

Description: This service returns COM_TRUE if <Flag> is set, otherwise it returns COM_FALSE.

Particularities: The flag is identified by the name <Flag>; this name is part of the service name as shown in the syntax description.[1)] The ISO 17356-4 implementation has to provide one *ReadFlag* service for each flag.

Return value:

FlagValue Value of the flag

3.9.3.3.2 ResetFlag

Service name: **`ResetFlag`**

Syntax: void ResetFlag_<Flag>()

Parameter (in): None

Parameter (out): None

Description: This service resets <Flag>.

Particularities: The flag is identified by the name <Flag>; this name is part of the service name as shown in the syntax description.[2)] The ISO 17356-4 implementation has to provide one *ResetFlag* service for each flag.

Status: None

3.9.3.4 Communication services

3.9.3.4.1 SendMessage

Service name: **`SendMessage`**

Syntax:
```
StatusType SendMessage (
    MessageIdentifier <Message>,
    ApplicationDataRef <DataRef>
    )
```

Parameter (in):

Message: Message identifier (C identifier)

DataRef: Reference to the application's message data to be transmitted

Parameter (out): None

Description: The service *SendMessage* updates the message object identified by <Message> with the application message referenced by the <DataRef> parameter.

---

1) For a given flag ABC, the name of the macro to read the flag is *ReadFlag_ABC()*.

2) For a given flag ABC, the name of the macro to reset the flag is *ResetFlag_ABC()*.

External communication:

If <Message> has the Triggered Transfer Property, the update is followed by immediate transmission of the I-PDU associated with the message, except when the message is packed into an I-PDU with Periodic Transmission Mode; in this case, no transmission is initiated by the call to this service.

If <Message> has the Pending Transfer Property, no transmission is caused by the update.

The service *SendMessage* resets all flags (Notification Classes 2 and 4) associated with <Message>.

Internal communication:

The message <Message> is routed to the receiving part of the IL.

Status:

Standard:

— This service returns E_OK if the service operation completed successfully.

Extended:

In addition to the standard status code defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it refers to a message that is received or to a dynamic-length or zero-length message.

3.9.3.4.2 ReceiveMessage

Service name: **`ReceiveMessage`**

Syntax:
```
StatusType ReceiveMessage (
    MessageIdentifier <Message>,
    ApplicationDataRef <DataRef>
    )
```

Parameter (in):

Message: Message identifier (C identifier)

Parameter (out):

DataRef: Reference to the application's message area in which to store the received data

Description: The service *ReceiveMessage* updates the application message referenced by <DataRef> with the data in the message object identified by <Message>. It resets all flags (Notification Classes 1 and 3) associated with <Message>.

Status:

Standard:

— This service returns E_OK if data in the queued or unqueued message identified by <Message> are available and returned to the application successfully.

— This service returns E_COM_NOMSG if the queued message identified by <Message> is empty.

— This service returns E_COM_LIMIT if an overflow of the message queue identified by <Message> occurred since the last call to ReceiveMessage for <Message>. E_COM_LIMIT indicates that at least one message has been discarded since the message queue filled. Nevertheless, the service is performed and a message is returned. The service ReceiveMessage clears the overflow condition for <Message>.

Extended:

In addition to the standard status codes defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it refers to message that is sent or to a dynamic-length or zero-length message.

3.9.3.4.3 SendDynamicMessage

Service name: **`SendDynamicMessage`**

Syntax:
```
StatusType SendDynamicMessage (
    MessageIdentifier <Message>,
    ApplicationDataRef <DataRef>,
    LengthRef <LengthRef>
    )
```

Parameter (in):

Message: Message identifier (C identifier)

DataRef: Reference to the application's message data to be transmitted

LengthRef: Reference to a value containing the length of the data in the message

Parameter (out): None

Description: The service *SendDynamicMessage* updates the message object identified by <Message> with the application data referenced by the <DataRef> parameter.

If <Message> has the Triggered Transfer Property, the update is followed by immediate transmission of the I-PDU associated with the message, except when the message is packed into an I-PDU with Periodic Transmission Mode; in this case, no transmission takes place.

If <Message> has the Pending Transfer Property, no transmission is caused by the update.

The service *SendDynamicMessage* resets all flags (Notification Classes 2 and 4) associated with <Message>.

Particularities: This service can be used with unqueued messages only. This service is provided for external communication only.

Status:

Standard:

— This service returns E_OK if the service operation completed successfully.

Extended:

In addition to the standard status code defined above, the following status codes are supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it refers to a received message, a static-length message or a zero-length message.

— This service returns E_COM_LENGTH if the value to which <LengthRef> points is not within the range 0 to the maximum length defined for <Message>.

3.9.3.4.4 ReceiveDynamicMessage

Service name: **`ReceiveDynamicMessage`**

Syntax:

```
StatusType ReceiveDynamicMessage (
    MessageIdentifier <Message>,
    ApplicationDataRef <DataRef>,
    LengthRef <LengthRef>
)
```

Parameter (in):

Message: Message identifier (C identifier)

Parameter (out):

DataRef: Reference to the application's message area in which to store the received data.

LengthRef: Reference to an application variable in which to store the message length.

Description: The service *ReceiveDynamicMessage* updates the application message referenced by <DataRef> with the data in the message object identified by <Message>. It resets all flags (Notification Classes 1 and 3) associated with <Message>.

The length of the received message data is placed in the variable referenced by <LengthRef>.

Particularities: This service can be used with unqueued messages only. This service is provided for external communication only.

Status:

Standard:

— This service returns E_OK if data in the unqueued message identified by <Message> is returned to the application successfully.

Extended:

In addition to the standard status code defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it refers to a message that is sent, a queued message, a static-length message or a zero-length message.

3.9.3.4.5 SendZeroMessage

Service name: **`SendZeroMessage`**

Syntax:
```
StatusType SendZeroMessage (
    MessageIdentifier <Message>
)
```

Parameter (in):

Message: Message identifier of the zero-length message (C identifier)

Parameter (out): None

Description:

External communication:

The service *SendZeroMessage* causes immediate transmission of the I-PDU associated with the zero-length message <Message> except when this message is associated with an I-PDU with Periodic Transmission Mode; in this case, no transmission is initiated by the call to this service.

The service *SendZeroMessage* resets all flags (Notification Classes 2 and 4) associated with <Message>.

Internal communication:

The message <Message> is routed to the receiving part of the IL for notification.

Status:

Standard:

— This service returns E_OK if the service operation completed successfully.

Extended:

In addition to the standard status code defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it refers to a non-zero-length message.

3.9.3.4.6 GetMessageStatus

Service name: **`GetMessageStatus`**

Syntax: StatusType GetMessageStatus (

MessageIdentifier <Message>

)

Parameter (in):

Message: Message identifier (C identifier)

Parameter (out): None

Description: The service *GetMessageStatus* returns the current status of the message object <Message>.

Status:

Standard:

— This service returns E_COM_NOMSG if the message queue identified by <Message> is empty.

— This service returns E_COM_LIMIT if an overflow of the message queue identified by <Message> occurred since the last call to ReceiveMessage for <Message>.

— This service returns E_OK if none of the conditions specified above is applicable or fulfilled and no error indication is present.

Extended:

In addition to the standard status codes defined above, the following status code is supported:

— This service returns E_COM_ID if the parameter <Message> is out of range or if it does not refer to a queued message.

3.9.3.4.7 COMErrorGetServiceId

Service name: **`COMErrorGetServiceId`**

Syntax: COMServiceIdType COMErrorGetServiceId (void)

Parameter (in): None

Parameter (out): None

Description: The service *COMErrorGetServiceId* (which may be implemented as a macro) returns the identifier of the ISO 17356-4 service where the error occurred.

Caveats: The service *COMErrorGetServiceId* shall only be called from *COMErrorHook*, otherwise the return value is undefined.

Return value: Service Identifier

3.9.3.4.8 COMError_Name1_Name2 macros

COMError_Name1_Name2 is the pattern for the names of macros which are used to access (from within the function *COMErrorHook*) parameters of the ISO 17356-4 service which called *COMErrorHook*.

The parts of the macro names are defined as follows:

— *COMError:* is a fixed prefix.

— *Name1*: is the name of the service, e.g. SendMessage.

— *Name2:* is the name of the parameter, e.g. DataRef.

3.9.4 Routines provided by the application

3.9.4.1 StartCOMExtension

Service name: **`StartCOMExtension`**

Syntax: StatusType StartCOMExtension (void)

Parameter (in): None

Parameter (out): None

Description: The routine *StartCOMExtension* is provided by the application and is called by the ISO 17356-4 implementation at the end of the *StartCOM* routine. It can be used to extend the start-up routine with initialization functions (e.g. *InitMessage*) or additional start-up functions (e.g. *StartPeriodic*).

Status:

Standard and Extended:

— This service returns E_OK if it completed successfully.

— This service returns an implementation-specific status code to indicate that an error occurred during its execution.

3.9.4.2 Callouts

Service name: **`COMCallout(CalloutRoutineName)`**

Syntax: COMCallout (CalloutRoutineName)

Parameter (in): None

Parameter (out): None

Description: The routine *CalloutRoutineName* is provided by the application and is called by the ISO 17356-4 implementation. It can be used to extend the ISO 17356-4 functionality with application-related functions (e.g. gatewaying).

The return value indicates whether the IL shall continue (COM_TRUE) or abandon (COM_FALSE) further processing of this message or I-PDU after the callout returns.

Return value: The routine *CalloutRoutineName* shall return a return value of the type CalloutReturnType. The return value contains information regarding whether or not to continue processing.

3.9.4.3 COMErrorHook

Service name: **COMErrorHook**

Syntax: void COMErrorHook (

StatusType <Error>

)

Parameter (in):

Error: Identifier of the occurred error

Parameter (out): None

Description: The service *COMErrorHook* is provided by the application and is called by COM at the end of a COM service which returns a status code not equal to E_OK.

Status: None

4 Minimum requirements of lower communication layers

This Clause describes the requirements of the lower communication layers that are used together with this part of ISO 17356. The lower layers could be the Network Layer or the Data Link Layer. The lower layers shall be capable of transmitting and receiving both fixed and dynamic-length I-PDUs as determined by the Interaction Layer. Therefore, the following three services are required:

— A *Request* service to pass control information and an I-PDU to the underlying layer and cause the I-PDU to be transmitted as soon as possible: The length of the I-PDU is mandatory control information for dynamic-length I-PDUs.

— A *Confirmation* service to confirm that a transmission of an I-PDU has been carried out: Status information shall be passed from the underlying layer to this part of ISO 17356. Depending on the outcome of the transmission, this status is either success or failure; in the case of a failure, the type of failure could be specified. The *Confirmation* service allows asynchronous behaviour between this part of ISO 17356 and the lower layer to be achieved.

— An *Indication* service to receive an I-PDU and pass status information from the underlying layer network to this part of ISO 17356: The length of the received I-PDU is mandatory status information for dynamic-length I-PDUs. Depending on the outcome of the reception, status also indicates either success or failure; in the case of a failure, the type of failure could be specified.

Additionally, the underlying layer shall be capable of broadcast transmission. If this is not the case, addressing more than one receiver on the same bus is not possible.

For the Controller Area Network (CAN) protocol, the Network Layer that is specified in ISO 15765-2 fulfils the above minimum requirements.

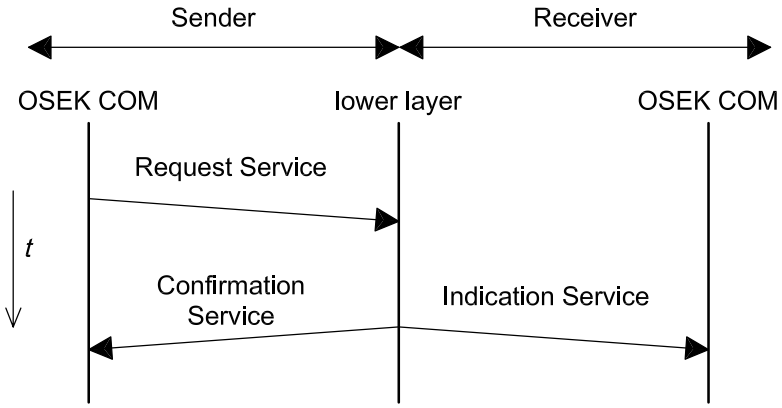


Figure 22 — Service calls required by COM but provided by a lower layer

5 Conformance Classes

Various application software requirements and specific system capabilities (e.g. communication hardware, processor and memory) require different levels of communication software functionality.

This part of ISO 17356 defines these levels as “Communication Conformance Classes” (CCCs). The main purpose of the conformance classes is to ensure that applications that have been built for a particular conformance class are portable across different ISO 17356-4 implementations and ECUs featuring that same level of conformance class. Hence, different implementations of the same CCC provide the same set of services and functionality to the application.

ISO 17356-4 implementation conforms to a CCC only if it provides all the features defined for that conformance class. However, system generation needs only to link those ISO 17356-4 services that are required for a specific application. A specific CCC is selected at system generation time.

This part of ISO 17356 defines the following CCCs:

CCCA:

CCCA defines the minimum features to support internal communication only; i.e. no support for external communication is available. Unqueued messages shall be supported. No message status information shall be supported in order to allow for a lean implementation of the communication kernel. The ISO 17356-4 services *StartCOM*, *StopCOM*, *GetCOMApplicationMode*, *InitMessage*, *SendMessage*, *ReceiveMessage*, *COMErrorGetServiceId*, the *COMError_Name1_Name2* macros and Notification Class 1 (except for the Flag notification mechanism) shall be supported.

CCCB:

CCCB defines features to support internal communication only; i.e. no support for external communication is available. All features of CCCA shall be supported with the following extensions: message status information (*GetMessageStatus* API service) and queued messages.

CCC0:

CCC0 defines minimum features to support internal and external communication. All features of CCCA shall be supported as well as Notification Class 2, byte order conversion and Direct Transmission Mode.

CCC1:

All features of this part of ISO 17356 shall be supported.

Table 3 — Definition of conformance classes

Features	CCCA	CCCB	CCC0	CCC1
Unqueued messages	√	√	√	√
Notification Class 1	√ [a]	√	√	√
Queued messages		√		√
Message status information		√		√
External communication			√	√
Triggered Transfer Property			√	√
Notification Class 2			√	√
Byte order conversion			√	√
Direct Transmission Mode			√	√
Filtering				√
Pending Transfer Property				√
Zero-length messages				√
Dynamic-length messages				√
Periodic Transmission Mode				√
Mixed Transmission Mode				√
Minimum delay time				√
Deadline Monitoring				√
Notification Class 3				√
Notification Class 4				√
Callouts				√

[a] Flag notification mechanism is not supported in CCCA.

Annex A
(informative)

Use of ISO 17356-4 (COM) with an OS not conforming to ISO 17356-3

This part of ISO 17356 can be implemented so that it works with operating systems other than the one described in ISO 17356-3. Such an implementation is simplified by the fact that only a limited amount of entities of ISO 17356-3 are used within this part of ISO 17356. To use this part of ISO 17356 with another operating system, the following facilities shall be offered by that operating system:

— tasks (basic and extended);

— events; and

— interrupt service routines (ISR) category 2.

Systems which can map these facilities such that they comply with their respective definition in ISO 17356-3 can fully support an ISO 17356-4 implementation.

Annex B
(informative)

Application notes

B.1 Zero-length messages

The main purpose of zero-length messages is to provide a signalling mechanism that is independent of the location of the sender and the receivers (locally inside one ECU or across the network) and to trigger a send request for an I-PDU containing messages configured as having the Pending Transfer Property.

When a zero-length message arrives, notification takes place. In the case of external transmission, notification is invoked upon the arrival of the containing I-PDU.

If an I-PDU is configured with the Direct Transmission Mode, a message configured with the *triggered* property is needed to request a transmission of the I-PDU. The *triggered* message can be a zero-length message.

Note that when an I-PDU contains more than one message with the Triggered Transfer Property, the receiver is not able to tell which message caused the I-PDU's transmission.

B.2 Use of callbacks

A callback is one of the notification mechanisms that can be invoked in response to an event in the IL. A callback with the name "cb1" would be declared in the application source as follows:

```
COMCallback(cb1)
{
…
}
```

When the declared event in the IL occurs, the IL calls the callback. This means that the context in which the callback is called (such as task priority if the IL is part of a task, or interrupt priority if the IL is part of an ISR) is determined by the implementation.

Because a callback is called as part of the IL, when the appropriate event occurs it gives the fastest response time to the arrival of a new message. However, because it runs as part of the IL, a callback can prevent the IL from being re-entered depending upon the implementation. Therefore, it can be necessary to ensure that the callback exits rapidly in order to prevent message loss.

B.3 m:n communication

The senders and receivers of a message are configured at system generation time.

On the **receiver** side, a message can have any number of receivers (even zero) in each ECU. The application is allowed to access any message object with multiple tasks or ISRs. The application has to ensure consistency, while reading from a queued message object with multiple tasks or ISRs.

On the **sender** side, a message can have any number of senders (even zero) but only in one ECU. A message can only be stored in up to one message object. For external communication, only one message object can be contained within one I-PDU. Therefore, multiple senders have to reside upon the same ECU.

A message can also be configured to have zero senders *and* zero receivers. This allows message space to be reserved in an I-PDU for future use.

Receivers cannot be configured for zero-length messages. However, a notification can still be generated. If the notification is a flag, then ResetFlag shall be used to reset the flag as the read API calls cannot be used on zero-length messages.

This part of ISO 17356 is written from the viewpoint of the application. It describes how tasks or ISRs acting as senders can route data to tasks or ISRs acting as receivers, and it describes the functionality behind the API functions used. With respect to the application, this part of ISO 17356 supports n:m communication.

When seen from inside, the IL is only concerned with message objects and not with senders or receivers. Message data is managed in sending message objects, and the data is sent either directly (internal communication) or via an underlying layer (external communication) to possibly more than one receiving message object. For the API and functionality of the IL, it is not relevant which tasks or ISRs access a message object. If the description of this part of ISO 17356 only focused on the point of view of message objects, the IL would be described to support 1:m communication. By including in ISO 17356-6 information about which tasks or ISRs access which message objects, more efficient implementations can be realized.

B.4 I-PDU transmission

The IL is responsible for requesting the transmission of an I-PDU by the underlying layer. For I-PDUs with Direct or Mixed Transmission Modes, a minimum delay time can be configured per I-PDU. The IL shall postpone further transmissions of a specific I-PDU if the minimum delay time of this I-PDU has not expired. The minimum delay time starts on confirmation of an I-PDU by the underlying layer. If no postponed request exists, an I-PDU transmission is requested by the schedule when using the Periodic or Mixed Transmission Modes.

Transmission of a direct or mixed I-PDU is also requested when a contained message with Triggered Transfer Property is sent.

Note that an I-PDU that is configured with the Direct Transmission Mode and that contains no messages with the Triggered Transfer Property is never transmitted.

B.5 I-PDU transmission modes

The Direct Transmission Mode is appropriate when the message's application data is to be sent quickly whenever an update occurs.

Periodically transmitted I-PDUs produce a bus load that is easy to model. When direct and mixed I-PDUs are taken into account, bus loading is more difficult to model. However, as the IL can limit the maximum rate at which direct and mixed I-PDUs can be transmitted, worst-case bus load calculations are still possible.

The reception of a periodically transmitted I-PDU does not imply that a task or ISR that sends messages using that I-PDU is still functioning correctly. Such detection might be performed by the task or ISR sending a message whose contents are changed each time the message is sent.

The Mixed Transmission Mode can be used to transmit important changes quickly outside the periodic time schedule.

B.6 Queued and unqueued messages

The following two figures illustrate the behaviour of a queued message.

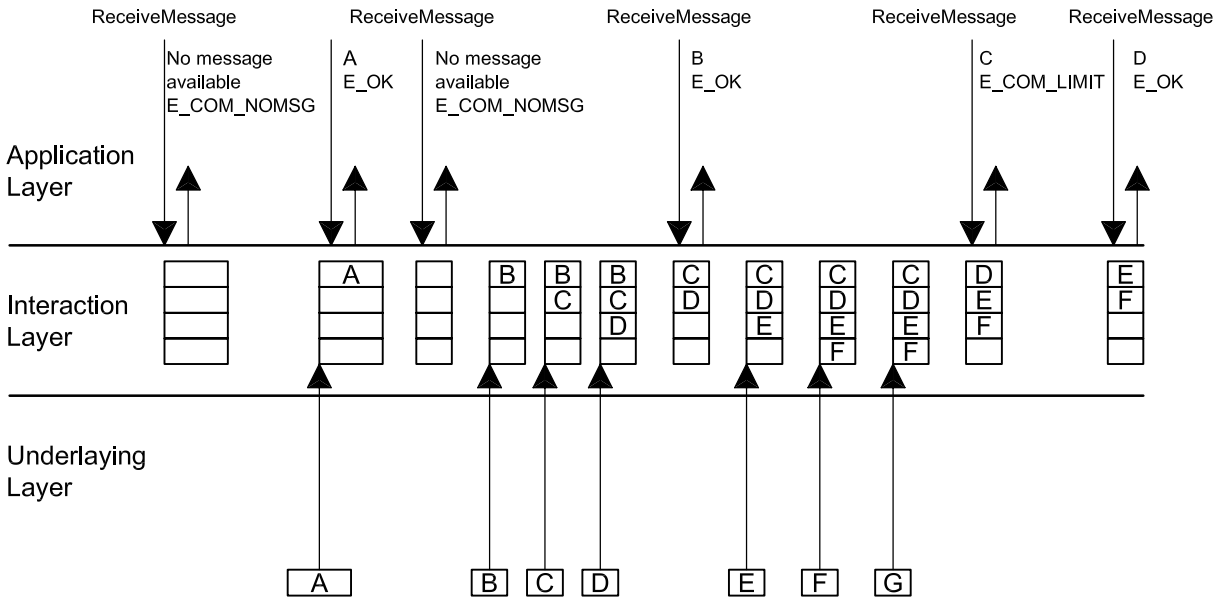


Figure B.1 —1 Behaviour of a queued message

Figure B.2 illustrates the behaviour of a queued message with a queue length of 1. In this case, once a message's data has been stored in the queue, no new message data can be stored until the old message has been consumed by the *ReceiveMessage* API service.

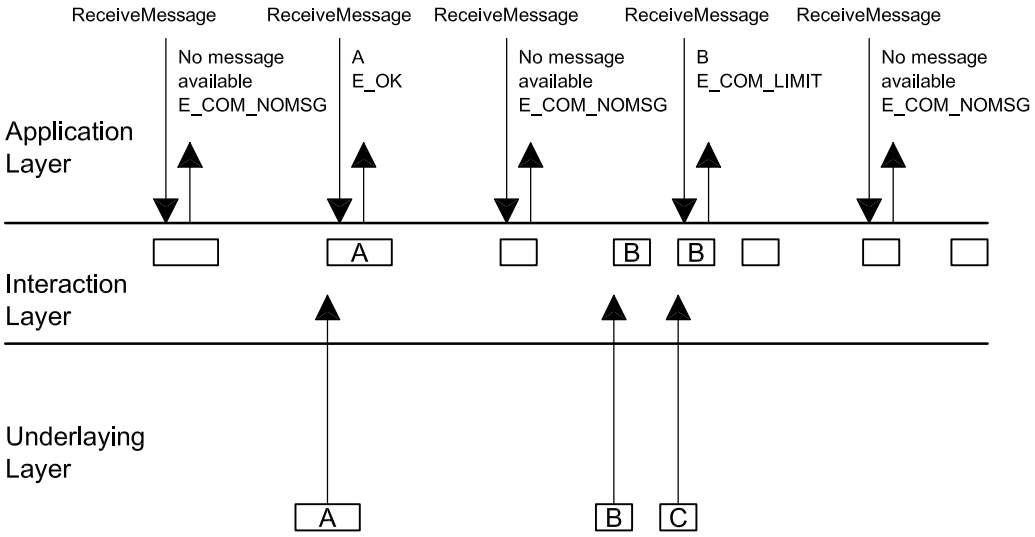


Figure B.2 — Behaviour of a queued message with a queue length of 1

Figure B.3 shows the behaviour of an unqueued message illustrating how the message data is overwritten each time a new message is received. Note that the behaviour of an unqueued message is not the same as that of a queued message with a queue length of 1.

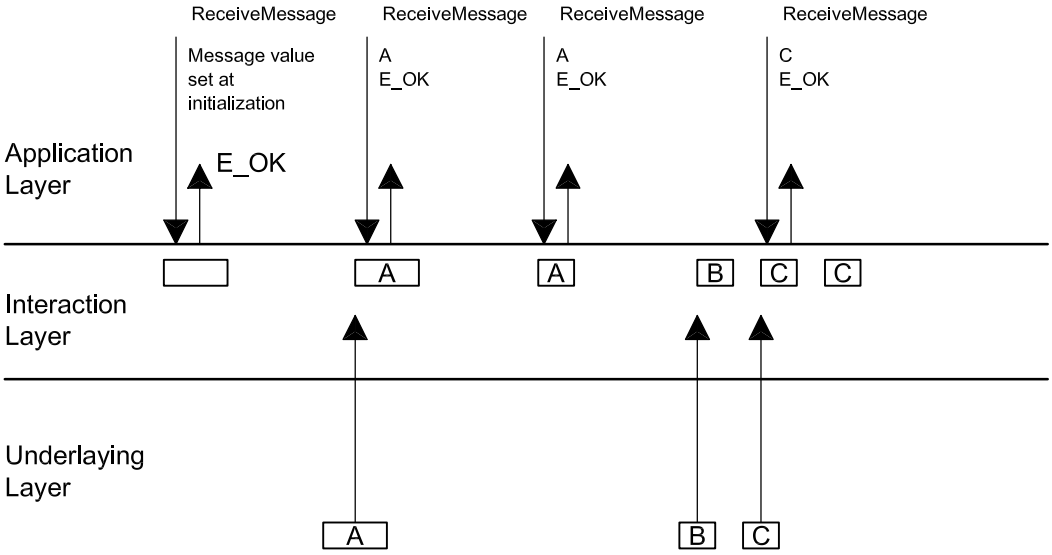


Figure B.3 — Behaviour of an unqueued message

B.7 Message data change detection

This annex is concerned with detecting changes in data between one instance of a message and another.

If an I-PDU containing a message is transmitted more often than the message is sent by the application, the situation can arise where a datum sent once by an application is received more than once by another. In some circumstances, this can be a problem in the receiver if it assumes that notification implies a new message, or can cause unnecessary CPU load due to the receiver being notified more often than necessary.

A number of techniques can be employed with this part of ISO 17356 to avoid incorrect multiple data reception.

The filter algorithms that permit the passage of messages with contents different from the previous message are the solution that is easiest to implement. These algorithms are *F_NewIsDifferent* and *F_MaskedNewDiffersMaskedOld*, although in specific cases others can achieve a similar purpose.

The filter algorithms are probably useful in a large majority of cases. However, there are some circumstances where they are not applicable. For example, where the datum is part of a structure that constitutes the message, or where it is necessary to have sequences of the same value.

Where a message is a structure, it is possible to add another element to the structure that is a sequence number. This sequence number does not need to have a large range; two values are adequate in many cases. When the structure is updated, the sequence number is incremented, modulo its range of values. At the receiving end, all occurrences of the message cause notification and the receiving task or ISR checks that the sequence number has moved on since the last reception. If it hasn't, then the message is a duplicate and can be discarded. If the sequence number has moved on, then it is a new message. By extending the range of sequence numbers, missing messages can also be detected, as they leave gaps in the sequence numbers. This solution can also be implemented using callouts as filters thereby avoiding application overheads in certain circumstances.

B.8 I-PDU transmission criteria

When considering external transmission, a message contained in an I-PDU can have an affect upon when the I-PDU is transmitted by the underlying layers as shown in the table below.

Table B.1 — I-PDU transmission criteria

		I-PDU transmission mode		
		Periodic	*Mixed*	*Direct*
Message Transfer Property	*Triggered*	The I-PDU is transmitted only with its declared period.	The I-PDU is transmitted with its declared period and also in response to a contained triggered message being sent.	The I-PDU is transmitted in response to this message being sent.
	Pending		The I-PDU is transmitted with its declared period, i.e. the I-PDU is not transmitted in response to this message being sent.	The I-PDU is not transmitted in response to this message being sent.

This table shows how a single message contained in an I-PDU affects the I-PDU's transmission. If there is more than one message in the I-PDU, then this table applies for each message in turn. For example, if an I-PDU is direct and contains a triggered message and a pending message, the I-PDU is only transmitted when the triggered message is sent.

In the case of internal messages, the data is placed in the receiver's message object as part of the send call. Therefore, internal communication can be regarded as synchronous.

B.9 Transfer modes for periodic transmissions

For messages that are assigned to I-PDUs which are configured to have the Periodic Transmission Mode, the configuration of the message's transfer property has no effect; the I-PDU is only transmitted at the points in time defined by its period. However, although the transfer mode is irrelevant in this special case, it is still advisable to assign the Pending Transfer Property to messages that are to be transmitted periodically.

One reason for this is that the application programmer usually defines the transfer property, but the transmission mode is usually defined by the person responsible for the overall network. Often, an I-PDU might have the Periodic Transmission Mode when the network design is started, but might later be reconfigured to have the Mixed Transmission Mode, e.g. by reassigning some other message to this I-PDU. If this happens, then the transfer property is again relevant and it should have been set to the correct value initially so as not to cause worry about correct transfer property at this later point in time.

B.10 Variable I-PDU Transmission Periods

Periodic I-PDUs have their periods fixed at system generation time. However, in certain circumstances, it is necessary to be able to give them different periods, after mode changes, for example. Although this part of ISO 17356 does not directly support variable period I-PDUs, they can be implemented using direct I-PDUs containing a triggered message.

The messages in the I-PDU that contain data would all be marked as pending and would be filled in by the application as appropriate. Transmission would be achieved by the application sending the triggered message. This might be sent by a task activated from an alarm specified in ISO 17356-3. By changing the alarm's period at run-time, the period of the I-PDU can also be changed. More complex schemes (for example, the task might implement a state machine) can result in arbitrarily complex I-PDU transmission patterns.

B.11 Interface to Indirect NM

The IL needs to call Indirect NM in order to indicate that a message has been transferred or that a message timeout has occurred (see 3.9.2). This is achieved by defining an NMCallback for a message in a monitored I-PDU. The message can be one that already exists in that I-PDU or a zero-length message used explicitly to cause an NMCallback.

Each implementation of Indirect NM might define different names for its *I_MessageTransfer.ind* and *I_MessageTimeOut.ind* routines. Therefore, the names used are configured as an NMCallback attribute of a message in the ISO 17356-6 file. Additionally, Indirect NM also needs to know which message caused the NMCallback. For this purpose, the NMCallback parameter called MonitoredIPDU uniquely identifies the message that caused the NMCallback. As a message can only appear in one I-PDU, and an I-PDU can only appear on one bus, this parameter is sufficient to identify the I-PDU and bus that caused the NMCallback. Therefore, the NMCallback indicates the condition of an I-PDU.

The values passed in the MonitoredIPDU parameter are defined per message in the ISO 17356-6 file. Therefore, a unique value can be chosen for each message.

B.12 Use of Overlapping Messages

This part of ISO 17356 allows messages in an I-PDU to overlap each other. One message may completely overlap another message or group of messages so that all of them are totally contained within the overlapping message. Alternatively, one message may only partially overlap another so that both have I-PDU bits in common and bits that are not in common.

Although overlapping messages have some uses, it is expected that they are unusual. Therefore, implementations should be designed to make the common case (non-overlapping messages) to be the most efficient.

Rules for message initialization apply equally to overlapping and non-overlapping messages. However, message initialization does not specify the order in which messages are initialized. Therefore, when initialization takes place as a result of initial values specified in the ISO 17356-6 file, the resulting message values in overlapping messages can differ between implementations. However, if *InitMessage* is used, message initialization can be written so that only relevant overlapping fields are set up, thereby improving portability.

When a system is configured, with or without overlapping messages, all the messages have the appropriate internal data structures generated, even though, e.g. in a particular ISO 17356-4 Application Mode, a message is not used. This is because message usage can depend upon information other than the ISO 17356-4 Application Mode. No special action is taken in the generation of this part of ISO 17356's internal data structures based upon whether or not messages overlap.

The rest of this section describes two possible uses for overlapping messages.

Overlapping messages can be useful when a group of signals need to be gatewayed from one network to another. If we assume that the message group occupies space in the I-PDU that has no messages not belonging to the group in it, then a single overlapping message can be used that encompasses all of the messages in the group. This means that the entire message group can be read from one I-PDU and written to another I-PDU simply by reading the single overlapping message. (This is similar to the way that structures work in C.)

A further use of overlapping messages is to allow the format of an I-PDU to be changed in response to, for example, the ISO 17356-4 Application Mode, or some tag field within the I-PDU. (This use is similar to unions in C.) In this case, any byte order conversion, filtering, copying to message objects and notification still take place for all the messages they are declared for, even if, in a certain mode, a message becomes irrelevant. This is because this part of ISO 17356 cannot selectively enable or disable messages based upon the ISO 17356-4 Application Mode. This implies that, under certain circumstances notifications are generated that are irrelevant. The application code shall be written so that it detects and correctly deals with these situations.

For example, a system has two messages, A and B, that have notifications that activate tasks TA and TB respectively when the message arrives. The messages are packed into the I-PDU so that they overlap. It is also assumed that message A is only relevant in ISO 17356-4 Application Mode X and message B is only relevant in ISO 17356-4 Application Mode Y.

When the system is initialized, the *StartCOMExtension* shall read the ISO 17356-4 Application Mode in order to decide whether to initialize message A or B.

When the system is running in ISO 17356-4 Application Mode X, reception of data in message A causes task TA to be activated. However, as these are in the same I-PDU task, TB is also activated.

As this is ISO 17356-4 Application Mode X rather than Y, TB's activation is undesirable but unavoidable. Therefore, the application shall be written in such a way that this problem is overcome. This can be achieved by the application code that receives the notification checking, whether or not the notification is acceptable in the current ISO 17356-4 Application Mode, and exiting if it is not. An outline of how this might be achieved is shown in the following code example.

```
TASK(TB) {
    if(GetCOMApplicationMode() != Y) {
        (void)TerminateTask();
    } else {
        /* we only get here if we are in the correct COM
         * application mode for this task
         */
        …
    }
}
```

Although this makes the task more complex, this only occurs in specific instances of overlapping message use rather than in the core of the IL.

Annex C
(informative)

Callouts

This annex describes some suggested uses for callouts.

Callouts provide a general mechanism to customize and enhance the behaviour of the IL. Callouts are configured statically, are invoked in response to the passage of a message or I-PDU and cannot be changed at run-time. The prototype for a callout allows it to return a value. This value is treated as a Boolean that can either prevent or allow further processing of the message or I-PDU.

Three uses of callouts are now described: custom filtering, gatewaying and replication. Each of these uses can apply equally well to I-PDUs or messages.

Declaration:

A callout is declared in the application code as follows:

```
COMCallout(co1)
{
…
}
```

This declares a callout called “co1”. As callouts have no parameters, it is best to have a callout for each separate use. This means that the callout implicitly knows which I-PDU or message it is dealing with.

Custom filtering:

The CPU-order message callouts can be used to implement custom filtering. When the callout is invoked, the message can be checked against some arbitrary criterion and the callout’s return value used to indicate whether or not the message passes the filter. Depending upon the callout's return value, the IL either discards the message or continues processing it.

For example, a custom filter might be implemented as follows:

```
COMCallout(filter1) {
   if(test criterion) {
      return COM_TRUE;
   } else {
      return COM_FALSE:
   }
}
```

so that the message is either discarded or passed based upon the test criterion.

Gatewaying:

In gatewaying, a message or I-PDU is received by the IL and then sent elsewhere, possibly to a different I-PDU on the same bus, or to an I-PDU on a different bus. When the callout is invoked, it copies the message or I-PDU to another I-PDU and then optionally initiates transfer of that I-PDU to the underlying layers, thereby causing its transmission on the bus. The return code from the callout can be used to indicate whether or not the message or I-PDU shall also be received by the controlling ECU.

Replication:

An ECU can interface to more than one bus. Therefore, it can be necessary to have the same I-PDU transmitted identically on more than one bus. This can be achieved with an I-PDU callout as it can place the contents of the outgoing I-PDU in some other I-PDU (destined for the same or another bus) and initiate its transmission if appropriate.

ICS 43.040.15

Price based on 55 pages