TECHNICAL SPECIFICATION

ISO/TS 17575-2

First edition
2010-06-15

Electronic fee collection — Application interface definition for autonomous systems —

Part 2: Communication and connection to the lower layers

Perception du télépéage — Définition de l'interface d'application pour les systèmes autonomes —

Partie 2: Communications et connexions aux couches plus basses

Reference number
ISO/TS 17575-2:2010(E)

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.

In other circumstances, particularly when there is an urgent market requirement for such documents, a technical committee may decide to publish other types of document:

— an ISO Publicly Available Specification (ISO/PAS) represents an agreement between technical experts in an ISO working group and is accepted for publication if it is approved by more than 50 % of the members of the parent committee casting a vote;

— an ISO Technical Specification (ISO/TS) represents an agreement between the members of a technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/PAS or ISO/TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/PAS or ISO/TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

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/TS 17575-2 was prepared by the European Committee for Standardization (CEN) Technical Committee CEN/TC 278, *Road transport and traffic telematics,* in collaboration with Technical Committee ISO/TC 204, *Intelligent transport systems*, in accordance with the Agreement on technical cooperation between ISO and CEN (Vienna Agreement).

ISO/TS 17575 consists of the following parts, under the general title *Electronic fee collection — Application interface definition for autonomous systems*:

— *Part 1: Charging*

— *Part 2: Communication and connection to the lower layers*

— *Part 3: Context data*

— *Part 4: Roaming*

Introduction

Autonomous systems

This part of ISO/TS 17575 is part of a series of specifications defining the information exchange between the Front End and the Back End in Electronic Fee Collection (EFC) based on autonomous on-board equipment (OBE). EFC systems automatically collect charging data for the use of road infrastructure including motorway tolls, zone-based fees in urban areas, tolls for special infrastructure like bridges and tunnels, distance-based charging and parking fees.

Autonomous OBE operates without relying on dedicated road-side infrastructure by employing wide-area technologies such as Global Navigation Satellite Systems (GNSS) and Cellular Communications Networks (CN). These EFC systems are referred to by a variety of names. Besides the terms autonomous systems and GNSS/CN systems, also the terms GPS/GSM systems, and wide-area charging systems are in use.

Autonomous systems use satellite positioning, often combined with additional sensor technologies such as gyroscopes, odometers and accelerometers, to localize the vehicle and to find its position on a map containing the charged geographic objects, such as charged roads or charged areas. From the charged objects, the vehicle characteristics, the time of day and other data that are relevant for describing road use, the tariff and ultimately the road usage fee are determined.

Some of the strengths of the autonomous approach to electronic fee collection are its flexibility, allowing the implementation of almost all conceivable charging principles, and its independence from local infrastructure, thereby predisposing this technology towards interoperability across charging systems and countries. Interoperability can only be achieved with clearly defined interfaces, which is the aim and justification of ISO/TS 17575.

Business architecture

This part of ISO/TS 17575 complies with the business architecture defined in the draft of the future International Standard ISO 17573. According to this architecture, the Toll Charger is the provider of the road infrastructure and, hence, the recipient of the road usage charges. The Toll Charger is the actor associated with the Toll Charging role. See Figure 1.

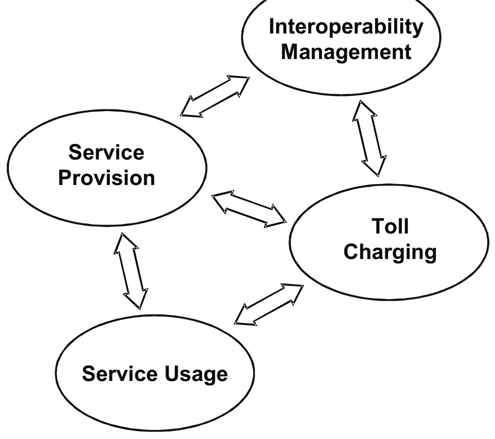

Figure 1 — The rolebased model underlying this Technical Specification

Service Providers issue OBE to the users of the road infrastructure. Service Providers are responsible for operating the OBE that will record the amount of road usage in all toll charging systems the vehicle passes through and for delivering the charging data to the individual Toll Chargers. In general, each Service Provider delivers charging data to several Toll Chargers, as well as each Toll Charger in general receives charging data from more than one Service Provider. Interoperability Management in Figure 1 comprises all specifications and activities that, in common, define and maintain a set of rules that govern the overall toll charging environment.

Technical architecture

The technical architecture of Figure 2 is independent of any particular practical realization. It reflects the fact that some processing functionalities can either be allocated to the OBE or to an associated off-board component (Proxy). An example of processing functionality that can be realized either on- or off-board is map-matching, where the vehicle locations in terms of measured coordinates from GNSS are associated to geographic objects on a map that either reside on- or off-board. Also tariffication can be done with OBE tariff tables and processing, or with an off-board component.

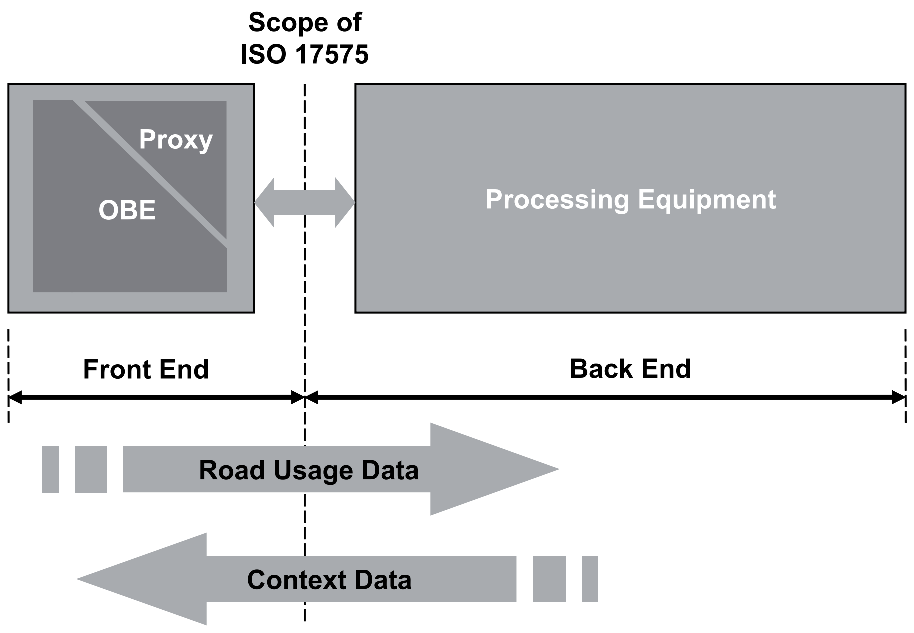

Figure 2 — Assumed technical architecture and interfaces

The combined functionality of OBE and Proxy is denoted as Front End. A Front End implementation where processing is predominately on OBE-side is known as a smart client (or intelligent client, fat client) or edge-heavy. A Front End where processing is mostly done off-board is denoted as thin-client or edge-light architecture. Many implementations between the “thin” and “thick” extremes are possible, as depicted by the gradual transition in the wedges in Figure 2. Both extremes of architectural choices have their merits and are one means where manufacturers compete with individual allocations of functionality between on-board and central resources.

Especially for thin client OBE, manufacturers might devise a wide variety of optimizations of the transfer of localization data between OBE and off-board components, where proprietary algorithms are used for data reduction and data compression. Standardization of this transfer is neither fully possible nor beneficial.

Location of the specification interface

In order to abstract from, and become independent of, these architectural implementation choices, the primary scope of ISO/TS 17575 is the data exchange between Front End and Back End (see the corresponding dotted line in Figure 2). For every toll regime, the Back End will send context data, i.e. a description of the toll regime in terms of charged objects, charging rules and, if required, the tariff scheme to the Front End, and will receive usage data from the Front End.

It has to be noted also that the distribution of tasks and responsibilities between Service Provider and Toll Charger will vary individually. Depending on the local legal situation, Toll Chargers will require “thinner” or “thicker” data, and might or might not leave certain data processing tasks to Service Providers. Hence, the data definitions in ISO/TS 17575 may be useful on several interfaces.

ISO/TS 17575 also provides for basic media-independent communication services that may be used for communication between Front End and Back End, which might be line-based or an air-link, and can also be used for the air-link between OBE and central communication server.

The parts of ISO/TS 17575

Part 1: Charging, defines the attributes for the transfer of usage data from the Front End to the Back End. The required attributes will differ from one Toll Charger to another, hence, attributes for all requirements are offered, ranging from attributes for raw localization data, for map-matched geographic objects and for completely priced toll transactions.

Part 2: Communication and connection to lower layers, defines basic communication services for data transfer over the OBE air-link or between Front End and Back End.

Part 3: Context Data, defines the data to be used for a description of individual charging systems in terms of charged geographical objects and charging and reporting rules. For every Toll Charger's system, attributes as defined in part 3 are used to transfer data to the Front End in order to instruct it which data to collect and report.

Part 4: Roaming, defines the functional details and data elements required to operate more than one EFC regime in parallel. The domains of these EFC regimes may or may not overlap. The charge rules of different overlapping EFC regimes can be linked, i.e. they may include rules that an area pricing scheme will not be charged if an overlapping toll road is used and already paid for.

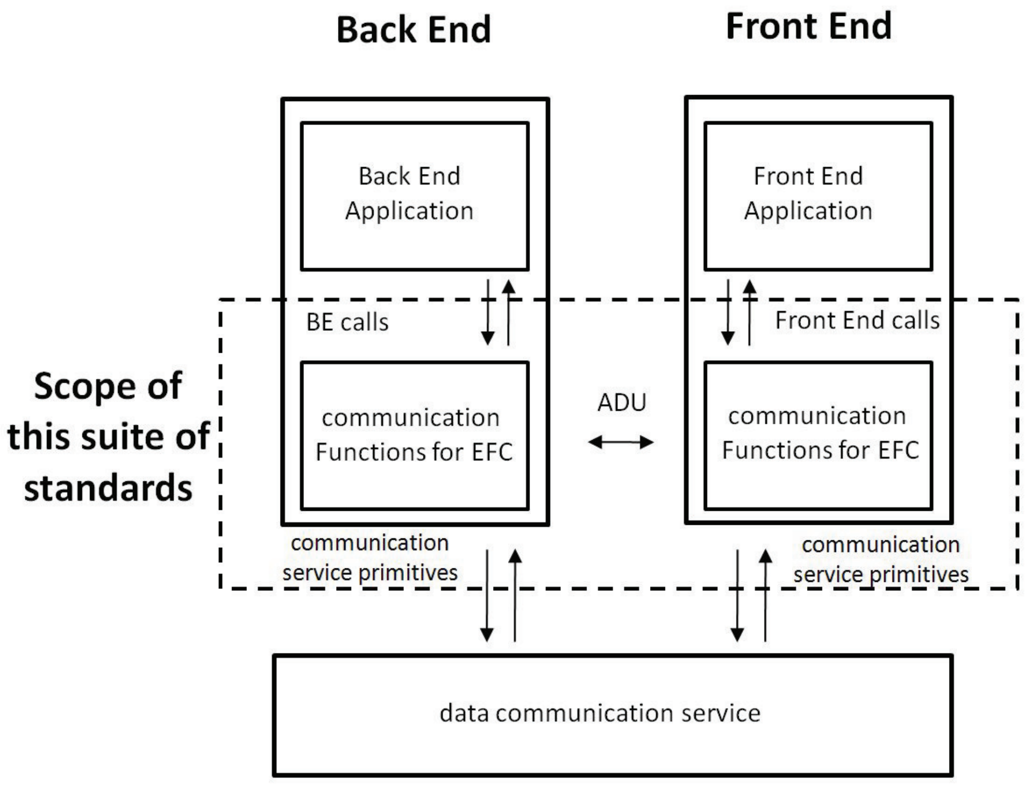
Back End
Front End
Back End Application
Front End Application
BE calls
Front End calls
Scope of this suite of standards
communication Functions for EFC
ADU
communication Functions for EFC
communication service primitives
communication service primitives
data communication service

Figure 3 — Scope of ISO/TS 17575

Applicatory needs covered by ISO/TS 17575

- The parts of ISO/TS 17575 are compliant with the architecture defined in the future International Standard ISO 17573.
- The parts of ISO/TS 17575 support charges for use of road sections (including bridges, tunnels, passes, etc.), passage of cordons (entry/exit) and use of infrastructure within an area (distance, time).
- The parts of ISO/TS 17575 support fee collection based on units of distance or duration, and based on occurrence of events.
- The parts of ISO/TS 17575 support modulation of fees by vehicle category, road category, time of usage and contract type (e.g. exempt vehicles, special tariff vehicles, etc.).
- The parts of ISO/TS 17575 support limiting of fees by a defined maximum per period of usage.
- The parts of ISO/TS 17575 support fees with different legal status (e.g. public tax, private toll).
- The parts of ISO/TS 17575 support differing requirements of different Toll Chargers, especially in terms of
  - geographic domain and context descriptions,
  - contents and frequency of charge reports,
  - feedback to the driver (e.g. green or red light), and
  - provision of additional detailed data on request, e.g. for settling of disputes.
- The parts of ISO/TS 17575 support overlapping geographic toll domains.
- The parts of ISO/TS 17575 support adaptations to changes in
  - tolled infrastructure,
  - tariffs, and
  - participating regimes.
- The parts of ISO/TS 17575 support the provision of trust guarantees by the Service Provider to the Toll Charger for the data originated from the Front End.

Electronic fee collection — Application interface definition for autonomous systems —

Part 2: **Communication and connection to the lower layers**

1 Scope

This part of ISO/TS 17575 defines how to convey all or parts of the data element structure defined in ISO/TS 17575-1 over any communication stack and media suitable for this application. It is focussed on mobile communication links. However, wired links shall use the same methodology.

To establish a link to a sequence of service calls initializing the communication channel, addressing the reception of the message and forwarding the payload are required. The required communication medium independent services are part of the definition of this part of ISO/TS 17575, represented by an abstract API.

The communication interface shall be implemented as an API in the programming environment of choice for the Front End (FE) system. The definition of this API in concrete terms is outside of the scope of this part of ISO/TS 17575. This part of ISO/TS 17575 specifies an abstract API that defines the semantics of the concrete API. An example concrete API is presented in Annex C. Where no distinction is made between the abstract and concrete communications APIs, the term “communications API” or just “API”, can be used.

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

ISO 14906:2004, *Road transport and traffic telematics — Electronic fee collection — Application interface definition for dedicated short-range communication*

3 Terms and definitions

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

3.1
attribute
application information formed by one or by a sequence of data elements, used for implementation of a transaction

3.2
authenticator
data appended to, or a cryptographic transformation of, a data unit that allows a recipient of the data unit to prove the source and/or the integrity of the data unit and protect against forgery

[ISO 14906:2004, definition 3.4]

3.3
Back End
generic name for the computing and communication facilities of the Service Provider and/or the Toll Charger

3.4
data element
datum which might itself consist of lower level data elements

3.5
data integrity
property that data has not been altered or destroyed in an unauthorized manner

[ISO 14906:2004, definition 3.10]

3.6
Front End
part(s) of the toll system where road usage data for an individual road user are collected, processed and delivered to the Back End

NOTE The Front End comprises the on-board equipment and an optional proxy.

3.7
Front End application
part of the Front End above the API

3.8
interoperability
ability of systems to provide services to and accept services from other systems and to use the services so exchanged to enable them to operate effectively together

3.9
on-board equipment
OBE
equipment located within or on the outside of the vehicle and supporting the information exchange across the interfaces of its sub-units, composed of the On Board Unit (OBU)

NOTE Other sub-units should be considered optional.

3.10
proxy
optional component part of the Front End that communicates with on-board equipment and processes road-usage data into a format compliant with this Technical Specification and delivers the data to the Back End

3.11
road
any stretch of land that can be navigated by a vehicle

3.12
service primitive (communication)
elementary communication service provided by the Application layer protocol to the application processes

[ISO 14906:2004, definition 3.16]

NOTE The invocation of a service primitive by an application process implicitly calls upon and uses services offered by the lower protocol layers.

3.13
service provider (toll)
legal entity providing its customers with toll services in one or more toll domains for one or more classes of vehicle

3.14
system
something of interest as a whole or as comprised of parts

NOTE A system can be referred to as an entity. A component of a system can itself be a system, in which case it can be called a subsystem.

3.15
tarrification
calculation of the tariff

3.16
toll
charge, tax, fee or duty in connection with using a vehicle within a toll domain

NOTE The definition is the generalization of the classic definition of a toll as a charge, a tax, or a duty for permission to pass a barrier or to proceed along a road, over a bridge, etc. The definition above also includes fees regarded as an (administrative) obligation, e.g. a tax or a duty.

3.17
toll charger
legal entity charging toll for vehicles in a toll domain

3.18
user
generic term used for the customer of a Toll Service Provider, one liable for toll, the owner of the vehicle, a fleet operator, a driver, etc. depending on the context

4 Abbreviations

For the purpose of this document, the following abbreviations apply unless otherwise specified.

— **ADU**	Application Data Unit (ISO 14906)
— **APDU**	Application Protocol Data Unit (ISO 14906)
— **AP**	Application Process (ISO 14906)
— **API**	Application programming interface
— **ASN.1**	Abstract Syntax Notation One (See ISO/IEC 8824-1.)
— **BE**	Back End
— **EFC**	Electronic Fee Collection (ISO 14906); here used equivalently to the term toll
— **EID**	Element Identifier (ISO 14906)
— **FE**	Front End
— **GNSS**	Global Navigation Satellite System
— **VAT**	Value added tax

5 EFC Front End communication architecture

5.1 General

The communications subsystem is required to establish the communication link between a Front End (FE) Application and a Back End (BE). It provides data transport for the tolling FE Application via the communications session that takes part across the line in Figure 4. For the case where a proxy is present in the Front End system the communications subsystem defines the communications between the BE and the Proxy. The link between the Proxy and the OBE is out of scope. For the case that no Proxy is present (the “Fat Client”) the communications subsystem defines the communications between the OBE and the BE.

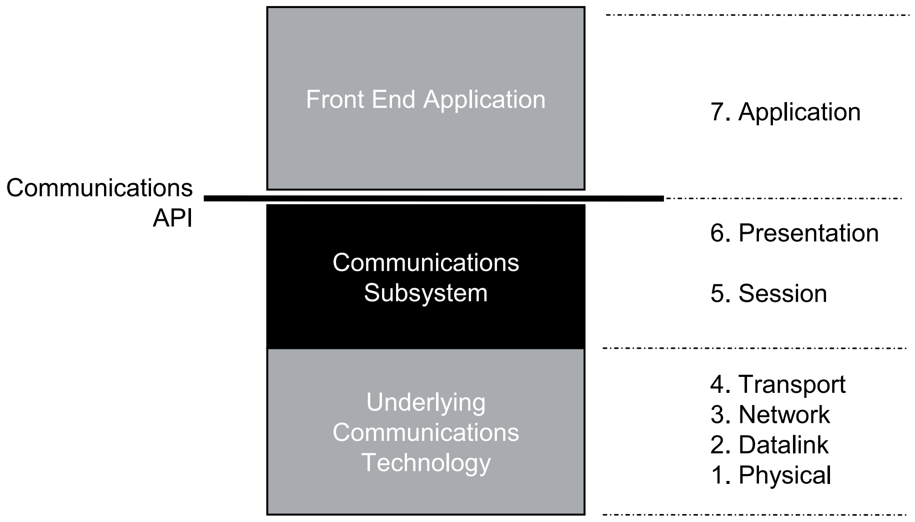

Figure 4 — Relationship between Application and Protocol Stack

The communications subsystem is further subdivided into two distinct components. The communications API itself offers communications functionality to the FE Application. Below this is the underlying communications technology which provides the functionality that the API abstracts. Although the API is independent of the underlying technology, it does place a number of functional demands upon it. For this reason the functional requirements on the underlying communications technology are listed in 8.2.

Some underlying technologies will be much more capable than others. For the case where a very capable technology is in use, the code interfacing the API to the underlying technology will serve little more function than a simple pass through. For more simplistic transport technologies the communications subsystem will have to do considerably more.

It is expected that these APIs will be “reflected” in the BE such that FEs and BEs can communicate over arbitrary bearer infrastructures. The specification of the BE API is outside the scope of this part of ISO/TS 17575.

5.2 Relations to the overall EFC architecture

The communications API provides the lower layers of the interface shown in Figure 5. The API has no semantic knowledge of the ADUs that it is carrying. It does differentiate between “standard specific” and “arbitrary” ADUs but it has no semantic knowledge about what these mean and simply carries them as transparent octet streams atop an arbitrary bearer that is selected at runtime.

6 Initialize transactions

6.1 General

The API carries two “types” of message (ADU). Structured elements relating directly to the definitions in ISO/TS 17575-1, and unstructured elements which are outside of the scope of this part of ISO/TS 17575 and receive no further consideration within it.

6.2 Addressing requested parts of a hierarchical data element structure

It is intended that the means and mechanisms of identifying data elements for transmission be defined in a projected part 3 of this Technical Specification.

6.3 Identifying payloads for transmission

It is intended that the means and mechanisms of identifying payloads for transmission be defined in a projected part 3 of this Technical Specification.

7 EFC communication services (functions)

7.1 General concept

The abstract API for the communications services can be implemented in any programming environment that defines the concept of event delivery, allowing the API to report information or to deliver results of operations to the Front End Application. The general sequence of events is:

— initialize and parameterize the communications interface;

— establish a communications session;

— transfer data in the context of the session;

— terminate the communications session;

— de-initialize the communications interface.

In a normal case the FE Application will initialize (a number of) communications interfaces when it first starts. An active session is then established either as a direct action by the FE Application or in response to an incoming request for a session from the BE. The flow of events through the lifetime depicted above is covered in the sections that follow and cross-referenced to the abstract API definition in Annex A.

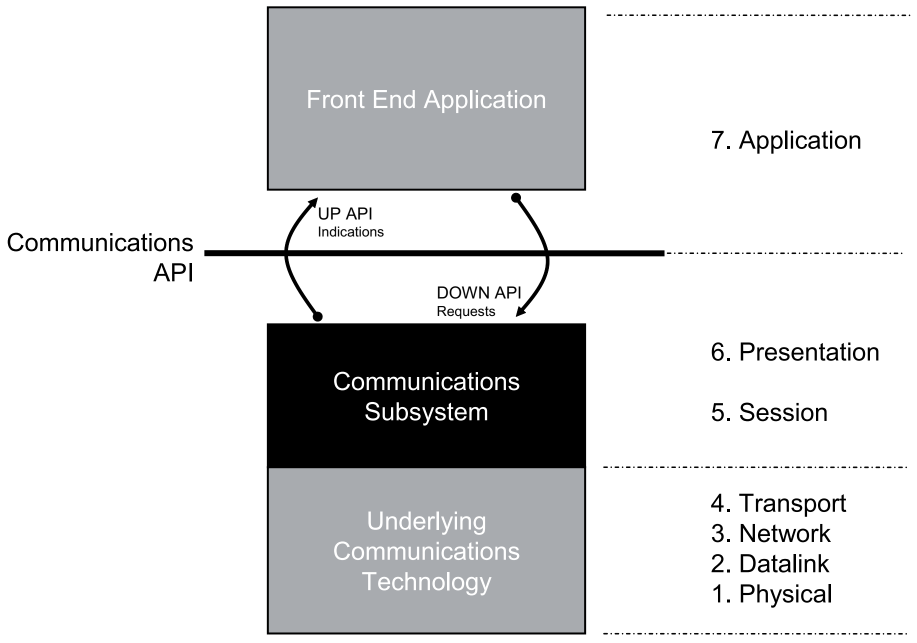

Figure 5 — Up/down interfaces

API calls down the communications stack, fall into two classes: synchronous and asynchronous. Synchronous calls giving results immediately are limited to those API calls which can quickly return a result (`InitialiseInstance` and `GetParameter`). Other API calls are asynchronous and return their results via the event mechanism which is initialized during `InitialiseInstance`.

NOTE This prevents threads from locking while waiting for information to be returned and so reduces the requirement for multithreaded programming, which is often inappropriate for embedded applications such as those which are typically seen in the field of application.

7.2 Initialization phase

7.2.1 General

The Front End Application shall initialize the communications interfaces that it wishes to make use of by means of calls to `InitialiseInstance`. For each instance created, the FE Application defines which underlying communications stack is to be used. More than one interface may be used at the same time and the choice between interfaces is the decision of the FE Application. The FE Application also provides a set of event reception capabilities which are used by the API to inform the FE Application of status changes.

Any additional parameterization of the instance that is required is performed by repeated calls to `SetParameter`. A set of parameters are recognised by the Communications API itself and those which are not are passed through transparently to the underlying stack. Queries for the existing state of parameters may be made via calls to `GetParameter`.

Once this process has been completed, the FE Application now has access to a (set of) communication instances. There will be delivered state changes for events that occur on any of these interfaces by means of `InstanceStateChange` event notification.

The state chart showing the relationship and interactions between each of these states is shown in Figure 6.

NOTE This chart only shows the states that are visible across the API.

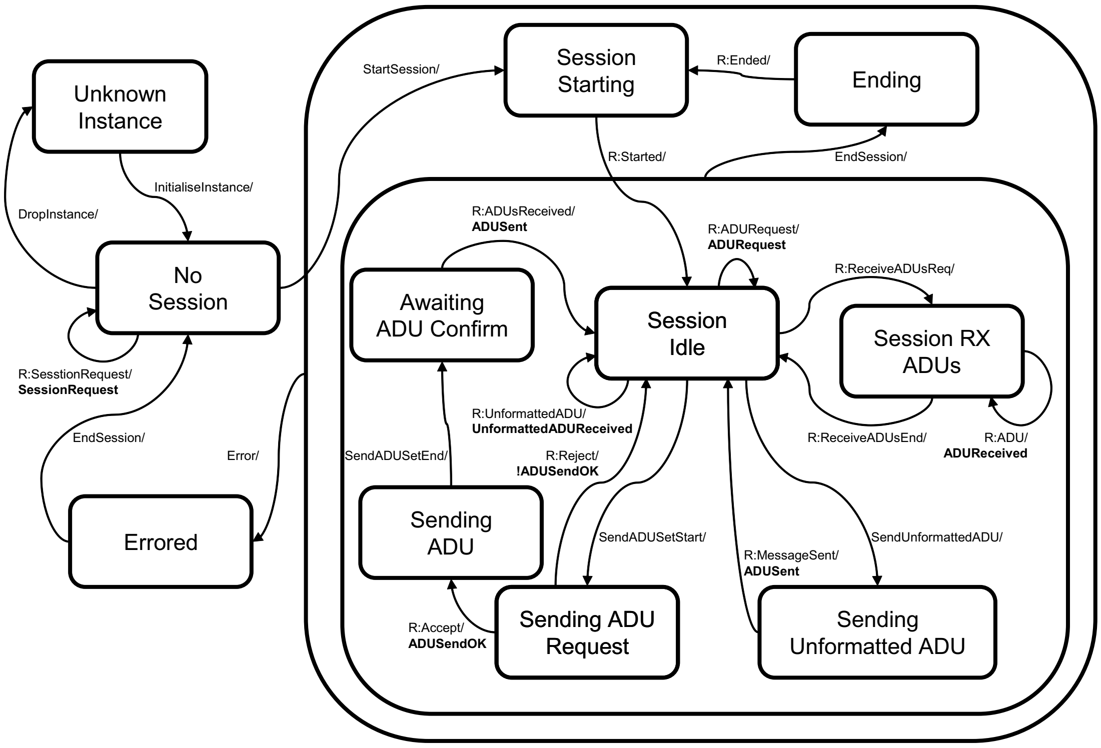

Figure 6 — Session State Chart (API visible states)

7.2.2 Incoming (BE to FE Application) Session Request

Optionally, the BE may wish to establish communications with the FE. In this case it performs whatever actions are appropriate for the particular communications technology concerned which will result in the FE Application being informed of the request at some point in time by means of a `SessionRequested` event with a datum `SessionHandle` indicating the identifier that the FE Application should use for the session. From this point onward the process is the same as for an outgoing session establishment in 7.2.3.

NOTE The FE Application might defer a session request for an arbitrary length of time, subject to operational constraints.

7.2.3 Outgoing (FE Application to BE) session establishment

The FE Application, either asynchronously of via the process in 7.2.2, requests a session within the chosen communication context by means of `StartSession`, parameterized with information about the session to be established. This returns immediately while also starting the process of creating the session within the communications technology layers below.

Once the session is established an `InstanceStateChange` event informs the FE Application of this fact. The session is now active and communications between the FE Application and the BE can take place within its context.

7.3 Point-to-point communication service primitives

7.3.1 General

Once the session is active then ADUs can be sent to the BE by the FE Application. Both structured (i.e. context aware) and unstructured (context unaware) communications capabilities are provided. All ADUs (structured and unstructured) are opaque to the communications layers and the distinction is only made to avoid the overhead of higher layer demultiplexing.

Within the context of the communication session the FE Application is considered the master and has control of the session.

7.3.2 Unstructured messages (ADUs)

The facility is provided to transit unformatted ADUs over the communications infrastructure. This facility is typically used for the purpose of software upgrades and various other manufacturer-specific information exchanges.

ADUs are sent via calls to `SendUnformattedADU`. The maximum size of ADU that can be sent via this mechanism is available via a parameter from the API. Once the message has been transmitted, an indication shall be provided (`ADUSent`) that it has been received successfully by the remote end and that further transactions are possible.

7.3.3 Structured messages (ADUs)

Structured ADUs are sent in sets. A set is constructed for transmission via a call to `SendADUSetStart`. This requests a permission to enter into structured ADU sending mode from the far end. ADUs are added to the set to be sent via calls to `SendADU` which returns the amount of space remaining in the transmit buffer. The set is closed with a call to `SendADUSetEnd`. Once transmission has been competed the FE Application is informed via an `ADUSent` event.

NOTE It is an implementation optimization option as ADUs are transmitted while the set is still being assembled. This means that the available buffer space indicated by the return from `SendADU` should be trusted more than local calculations in the FE Application as more space can be made available when elements are transmitted.

7.4 Session ending

When it is time for a session to end, the FE Application shall make a call to `EndSession` providing a suitable reason code. This will start the process of ending the session within the underlying communications technology. Session termination may not be immediate and transactions that are in process will be completed before closure occurs. FE Application implementers should expect to have to handle activities from the open session until an `InstanceStateChange` to state `STNoSession` is received.

7.5 Session failure

A session may end through no fault of the BE or the FE Application by means of intervening communications infrastructure failure. In this instance the FE Application shall be informed of the fact by means of an `InstanceStateChange` to `STErrored`. In this circumstance any ADUs sent by the FE Application for which it has not received an `ADUSent` confirmation should be assumed to have failed.

The communications technology shall not attempt to automatically re-establish the session. That is the responsibility of the FE Application. If the communications session was as a result of the BE request then the session should be re-established at the first convenient opportunity. If the session was as a result of an FE Application event then the decision to re-establish is left to the FE Application.

Note that some communications technologies are, by their nature, intermittently available. To support these a `StackAvail` call is provided to allow the FE Application to make intelligent choices between available infrastructures. If a medium fails during a session this shall be indicated according to the processes noted above.

7.6 Security considerations

All communications are encrypted. Certificate handling and encryption mechanisms are outside of the scope of this part of ISO/TS 17575.

7.7 Media selection options

Media may be selected between means of having several instances instantiated with independent communication technologies. The capabilities of each medium can be determined via `GetParameter` calls, and the local availability of any specific medium in an active instance can be determined through calls to `StackAvail`.

8 The use of a communication stack

8.1 General

This part of ISO/TS 17575 allows for multiple, concurrent, communications technologies to be used at the same time, and for the FE Application to be able to select amongst them the one most appropriate for the specific communication to take place. It is envisaged that this capability will be useful for “multi-mode” OBEs which can use different technologies depending upon the urgency of the communication, their capability and the extent of the infrastructures that are available.

There are, however, a number of underlying capabilities that any communication technology (or rather, stack which sits on top of the technology) needs to provide.

8.2 Requirements on a underlying communication technology

This part of ISO/TS 17575 is open for a wide range of underlying communications technologies. However, the following list of properties of these stacks shall be provided.

- Must be capable of supporting or emulating a “session”.
- Must be capable of reliably transferring ADUs, in sequence, bidirectionally across the link.
- Must be capable of reporting the safe receipt of ADUs at the remote end of the link.
- Must be capable of transporting data elements of arbitrary length between the FE and the BE.
- Must be capable of establishing a session from the FE to the BE.
- Must offer some means of signalling a demand from the BE to the FE Application that the BE wishes a session to be established.
- Must be capable of reliably detecting the loss of a session and of delivering that information to the communications API.
- Must deliver ADUs across the link in a timely fashion.
- Must be capable of carrying an appropriate amount of data.

8.3 Mobile terminated calls

The approach taken within this part of ISO/TS 17575 never requires an incoming, in band, communication port to be open. If a BE wishes to make a connection to an FE Application for any purpose, it can use out of band signalling to achieve that and the mechanisms used are outside the scope of this part of ISO/TS 17575. The range of out of band options is considerable (SMS messages, push button on the unit, broadcast signal, etc.) and makes the FE Application more secure as a result.

NOTE A consequence of this approach is that there is never any requirement for an IP addressable endpoint for a mobile terminated call. This avoids the security issues discussed above but also simplifies issues of Network Address Translation and private subnets, since the FE Application will only ever need to make an outgoing connection.

Annex A
(normative)

Abstract API definition

A.1 General

The communications API consists of a “down” API, from the Front End Application to the communications stack, and an “up” API, from the communications stack to the Front End Application. Each will be considered in turn.

A.2 Down API (Front End Application to communications stack)

A.2.1 InitialiseInstance

Purpose: Initializes the communications API for use. The interface is initialized in `STNoSession` condition.

Takes:

`StackID`	Underlying communications stack to be employed.
`Callbacks`	Reference to the event handlers to be used for this instance, refer to A.2 for details of these.

Precondition: The interface must not have been previously initialized.

Returns: A handle to the instance of the API. This handle is used for all communications.

Errors: Returns an invalid instance if it was not possible to create the instance.

A.2.2 SetParameter

Purpose: Set a parameter for this instance of the API. All parameters and values are expressed as strings.

Takes:

`Instance`	The instance to which this parameter relates.
`Parameter`	The parameter to be set.
`value`	The value to give to the parameter.

Precondition: A valid instance is required.

Returns: Error Code.

Errors: `ERNoError, ERNotSet`

A.2.3 GetParameter

Purpose: Get a parameter value for this instance of the API.

Takes:

`Instance`	The instance to which this parameter relates.
`Parameter`	The parameter to be set.

Precondition: A valid instance is required.

Returns: The specified parameter value as a string.

Errors: Invalid string if the parameter is not known.

A.2.4 DeleteParameter

Purpose: Delete a parameter from this instance of the API.

Takes:

`Instance`	The instance to which this parameter relates.
`Parameter`	The parameter to be deleted.

Precondition: A valid instance is required.

Precondition: Error Code.

Errors: `ERNoError`, `ERNotSet`

A.2.5 StackAvail

Purpose: Indicate if the communications stack is currently available.

Takes:

`Instance`	The instance to which this check relates.

Precondition: A valid instance is required.

Returns: Boolean indicating if the stack is currently available.

Errors: Will return FALSE in the case of an error.

A.2.6 DropInstance

Purpose: Delete an API interface

Takes:

`Instance`	The instance to which this drop request relates.
`Severity`	Speed with which this interface should be removed. (`SENormal`, `SEUrgent`, `SEUnconditional`)

Precondition: A valid instance in the `STNoSession` state (for `SENormal` and `SEUrgent`) is required.

Returns: Error code.

Errors: `ERUnknownInstance`, `ERBadState`, `ERNoError`

A.2.7 StartSession

Purpose: Start the process to establish a session in the context of this interface (i.e. using the established communications channel and parameters). Session establishment is defined by an event indicating a change to `STSessionIdle` when the session is in place. Parameterization of the session is via the `SetParameter`/`GetParameter` methods.

Takes:

`Instance`	The instance within which this session request should be met.
`Reason`	Reason for this session establishment.
`SessionHandle`	Any session handle information that is required.

Precondition: Active instance and no active session already. Correct parameterization in place.

Returns: Error code.

Errors: `ERInSession`, `ERNoInstance`, `ERUnknownEndpoint`, `ERInSession`, `ERSessionFailed`, `ERNoError`

A.2.8 EndSession

Purpose:	To start the process to end an active session in the context of this instance. Session end is confirmed via the state change event to state `STNoSession`. Under normal conditions any outstanding transactions in the context of the session will be completed before the close is performed.	
Takes:	`Instance`	The instance within which this session is to be terminated.
	`Reason`	The reason for this session termination.
Precondition:	Existence of an active session.	
Returns:	Error Code	
Errors:	`ERInSession, ERNoInstance`, `ERNoError`	

A.2.9 SendUnformattedADU

Purpose:	To send an unformatted ADU to the other end of the communication link. In the context of this document an unformatted ADU is one that has no special meaning to the communications API and is processed by vendor specific code. Confirmation of receipt at the far end is by means of an event indication.	
Takes:	`Instance`	The instance within which this message is to be sent.
	`MessageLen`	The length of the message, in octets, to be sent.
	`Message`	The message itself.
Precondition:	Existence of an active session with no transaction in progress.	
Returns:	Error Code	
Errors:	`ERBadState, ERNoInstance`, `ERSessionFailed`, `ERNoError`	

A.2.10 SendADUSetStart

Purpose:	To send a request to start sending a set of structured ADUs to the remote end of the communication link. Confirmation of acceptance of transfer is by means of an event `ADUSendOK`.	
Takes:	`Instance`	The instance within which this message is to be sent.
Precondition:	Existence of an active session with no transaction in progress.	
Returns:	Error code	
Errors:	`ERNoInstance`, `ERBadState`, `ERSessionFailed`, `ERNoError`	

A.2.11 SendADU

Purpose:	To add an ADU to the set to be sent to the remote end of the communication link.	
Takes:	`Instance`	The instance within which this message is to be sent.
	`ElementLen`	The length of the element, in octets, to be sent.
	`Element`	The element itself.
Precondition:	An active session exists with no transaction in progress and an element set is open for construction (See A.2.10).	
Returns:	Number of octets remaining in the transmission buffer.	
Errors:	Returns 0 in case of error.	

A.2.12 SendADUSetEnd

Purpose:	To conclude the set of structured ADUs (as defined in ISO/TS 17575-1) to be sent to the remote end of the communication link. Far end reception of the elements is confirmed by means of the event `ADUSent`.
Takes:	`Instance` The instance within which this message is to be sent.
Precondition:	An active session exists with no transaction in progress and an element set is open for construction (See A.2.10).
Returns:	Error Code
Errors:	`ERBadState`, `ERSessionFailed`, `ERNoError`

A.2.13 CommsQuery

Purpose:	To poll for the current state of the communications instance.
Takes:	`Instance` The instance to which this poll request relates.
Precondition:	A valid instance is required.
Returns:	The current state of the session;

`STUnknownInstance`	The instance is invalid
`STNoSession`	No session is in progress
`STStarting`	The session is starting
`STSessionIdle`	Session open and idle
`STSendingADU`	Sending ADUs
`STSendingADURequest`	Requesting permission to send ADUs
`STSendingUnformattedADU`	Sending an unformatted ADU
`STAwaitingADUConfirm`	Awaiting confirmation of ADU reception
`STSessionRxADU`	Receiving ADUs from far end
`STErrored`	In error state
`STEnding`	In the process of ending

Errors: `STUnknownInstance`

NOTE The error `STUnknownInstance` is reported as a state.

A.3 Up API (Communications Stack to Front End Application)

A.3.1 InstanceStateChange

Purpose:	To indicate to the FE Application that a change has occurred in the state of a communications instance	
Receives:	`Instance`	The instance to which this indication relates.
	`OldState`	The old state of the instance (as per A.2.13).
	`NewState`	The new state of the instance (as per A.2.13).

A.3.2 UnformattedADUReceived

Purpose: To indicate to the FE Application that an unformatted ADU has been received. After reception of this event the BE will automatically be informed that the FE Application has received the ADU satisfactorily.

Receives:

`Instance`	The instance to which this indication relates.
`UnformattedMessageLen`	The length of the incoming message.
`UnformattedMessage`	The message itself.

A.3.3 ADURequest

Purpose: To indicate to the FE Application that the BE is requesting specific element(s) from it. The FE Application should start the processes required to send an ADU set to deliver the requested elements to the BE.

Receives:

`Instance`	The instance to which this indication relates.
`Elements`	Elements to be returned to the BE.

A.3.4 ADUReceived

Purpose: To deliver to the FE Application an element from the BE structured in accordance with ISO/TS 17575-1. After reception of this event the BE will automatically be informed that the FE Application has received the ADU satisfactorily.

NOTE If several elements are combined into a single communications layer message then an indication of successful receipt will only be sent when the final element is delivered. This could result in re-transmission of some elements when communications are lost.

Receives:

`Instance`	The instance to which this indication relates.
`Element`	The element received.

A.3.5 ADUSent

Purpose: To confirm to the FE Application that an ADU or ADU set has been successfully received by the BE.

Receives:

`Instance`	The instance to which this indication relates.

A.3.6 ADUSendOK

Purpose: To confirm to the FE Application that it may now proceed to send a set of elements.

Receives:

`Instance`	The instance to which this indication relates.
`CanSend`	Flag indicating if send request was accepted.

A.3.7 SessionRequest

Purpose: To alert the FE Application that this instance has detected a session request. Parameters for the session may be claimed via `GetParameter`.

Receives:

`Instance`	The instance that generated this request.
`Handle`	The handle to be used for establishing the session.

Annex B
(normative)

PICS proforma

B.1 Introduction

This annex contains the Protocol Implementation Conformance Statements (PICS) proforma to be used for OBE implementation of the communication services as defined in Clauses 6, 7 and 8.

NOTE Media selection policies are outside the scope of this part of ISO/TS 17575.

B.2 Transactions support

B.2.1 General

This clause applies to implementations for Front End Communication APIs. The appropriate Communication Services for the BE shall use the same statements with the view from the BE.

B.2.2 Support of the down API

Table B.1 — Support of the down API

Description	Dispensation
API supports `InitialiseInstance`	Yes/No
`StackID` is supported	Yes/No
`Instance handle` will be provided	Yes/No
Invalid Instance returned when not possible to create an instance	Yes/No
API supports `SetParameter`	Yes/No
`Instance` is applied	Yes/No
`ERNoError` returned on successful parameter setting	Yes/No
`ERNotSet` returned on failure to set parameter	Yes/No
Maximum length of parameter handles	Number
Maximum length of value strings	Number
Parameter is stored following call	Yes/No
API supports `GetParameter`	Yes/No
`Instance` is applied	Yes/No
`Parameter` is applied	Yes/No
`Value` is returned as a string when input parameters are valid	Yes/No
An invalid string is returned when input parameters are not valid	Yes/No
API supports `DeleteParameter`	Yes/No
`Instance` is applied	Yes/No
`Parameter` is applied	Yes/No

Table B.1 (*continued*)

Description	**Dispensation**
`ERNoError` is returned when parameter is successfully deleted	Yes/No
`ERNotSet` is returned when parameter is not successfully deleted	Yes/No
`ERNotSet` is returned when Instance is not valid	Yes/No
`ERNotSet` is returned when parameter is not found/invalid	Yes/No
API supports DropInstance	Yes/No
`Instance` is applied	Yes/No
`Severity` is considered	Yes/No
`ERUnknown` instance is returned when an unknown instance is provided	Yes/No
`ERBadState` is returned when `Severity` is not `SEUnconditional` and not in `STNoSession` State	Yes/No
`ERNoError` is returned when instance is successfully dropped	Yes/No
API supports `StartSession`	Yes/No
`Instance` is applied	Yes/No
`Reason` is applied	Yes/No
`SessionHandle` is applied	Yes/No
`ERInSession` is returned if already in a session	Yes/No
`ERNoInstance` is returned if the instance is invalid	Yes/No
`ERUnknownEndpoint` is returned if the parameterised endpoint is not known	Yes/No
`ERSessionFailed` is returned if session establishment fails immediately	Yes/No
`ERNoError` is returned if session establishment starts correctly	Yes/No
API supports `EndSession`	Yes/No
`Instance` is applied	Yes/No
`Reason` is applied	Yes/No
`ERInSession` is returned if the session is not in `STSessionIdle` and `Reason` is not `RERemoteDrop`	Yes/No
`ERNoInstance` is returned if the instance is invalid	Yes/No
`ERNoError` is returned if the `EndSession` starts correctly	Yes/No
API supports `SendUnformattedADU`	Yes/No
`Instance` is applied	Yes/No
`MessageLen` is applied	Yes/No
`Message` is considered	Yes/No
`ERBadState` is returned if the session is not in the state `STSessionIdle`	Yes/No
`ERNoInstance` is returned if the instance is invalid	Yes/No
`ERSessionFailed` is returned if the session has failed	Yes/No
`ERNoError` is returned if progress is normal	Yes/No

Table B.1 (*continued*)

Description	Dispensation
API supports `SendADUSetStart`	Yes/No
`Instance` is supported	Yes/No
`ERNoInstance` is returned if the instance is invalid	Yes/No
`ERBadState` is returned if the instance is not in the state `STSessionIdle`	Yes/No
`ERSessionFailed` is returned if the session has failed	Yes/No
`ERNoError` is returned if the progress is normal	Yes/No
API supports `SendADU`	Yes/No
`Instance` is applied	Yes/No
`ElementLen` is considered	Yes/No
`Element` is considered	Yes/No
`0` is returned in case of error	Yes/No
Remaining space in the transmit buffer is returned in case of no error	Yes/No
API supports `SendADUSetEnd`	Yes/No
`Instance` is considered	Yes/No
`ERBadState` is returned if the session is not in the state `STSendingElements`	Yes/No
`ERSessionFailed` is returned if the session has failed	Yes/No
`ERNoError` is returned if the progress is normal	Yes/No
API supports `CommsQuery`	Yes/No
`Instance` is considered	Yes/No
`STUnknownInstance` is returned if the instance is unknown	Yes/No
The current state of the instance is returned in normal progress	Yes/No
API supports `StackAvail`	Yes/No
`Instance` is considered	Yes/No
`FALSE` is returned if the instance is invalid	Yes/No
`FALSE` is returned if the stack is not available	Yes/No
`TRUE` is returned if the stack is available	Yes/No

B.2.3 Support of the Up API

Table B.2 — Support of the Up API

Description	Dispensation
API supports `InstanceStateChange` Event	Yes/No
`Instance` is delivered	Yes/No
`OldState` is delivered	Yes/No
`NewState` is delivered	Yes/No
Event is generated whenever an externally visible state change is generated	Yes/No
API supports `UnformattedADUReceived` Event	Yes/No
`Instance` is delivered	Yes/No
`UnformattedMessageLen` is delivered	Yes/No

Table B.2 (*continued*)

Description	Dispensation
`UnformattedMessage` is delivered	Yes/No
Event is generated whenever an `UnformattedMessage` is received	Yes/No
API supports `ADUReceived` Event	Yes/No
`Instance` is delivered	Yes/No
`Element` is delivered	Yes/No
Event is generated whenever an element is received	Yes/No
API supports `ADUSent` Event	Yes/No
`Instance` is delivered	Yes/No
Event is generated whenever a message has been acknowledged by the far end	Yes/No
API supports `ADUSendOK` Event	Yes/No
`Instance` is delivered	Yes/No
`CanSend` is delivered	Yes/No
Event is generated whenever far end has indicated its ability to receive elements	Yes/No
`CanSend` is `TRUE` when far end is able to receive elements, `FALSE` otherwise	Yes/No
API supports `SessionRequest` Event	Yes/No
`Instance` is delivered	Yes/No
`Handle` is delivered	Yes/No
Event is generated when a stimulus is detected to indicate remote end wants to establish a session	Yes/No

B.3 Use of communication stacks

B.3.1 Supported communication stacks

This clause applies to both Front End and Back End (BE).

Table B.3 — Supported communication stacks

Description	Value(s)
Communications supports the use of TCP/IP v4	Yes/No
Communication supports the use of TCP/IP v6	Yes/No
Communication originates on IP port number	Any/Specify
Communication terminates on IP port number	Specify
The Front End accepts fully qualified domain names as end point	Yes/No
The Front End closes logical session automatically after	Specify in seconds

B.4 Front End Storage capacity

NOTE This clause only applies to Front Ends.

B.4.1 Storage capacity for modules and contact details

Table B.4 — Storage capacity for modules and contact details

Description	Maximum value or range
Maximum single message size	
Maximum number of instances communication sessions in parallel	
Maximum storage available for queuing of messages	
Maximum number of keys/trust certificates available	
Maximum number of elements that can be transmitted in a single transaction	

B.4.2 Generic values

Table B.5 — Generic values

Description	**Max**mum **value or range**
Integers (minimum and maximum value)	
Strings (maximum size)	
Parameter maximum length	
Parameter value maximum length	

B.4.3 Security of communication

Table B.6 — Security of communication

Description	Dispensation
All communication is over encrypted channels	Yes/No
Specify key length for encryption	Number
Specify encryption technology used	Text
Symmetric or Asymmetric keys?	Symmetric/Asymmetric

Annex C
(informative)

API requirements

C.1 Introduction

C.1.1 General

In this annex the requirements that drive the definition of the communications API are outlined.

C.1.2 Non-functional requirements

- The communications subsystem is required, in a standards-oriented manner, to address how a link is established, data transferred over it and cleared at the end of the transaction.
- The communications subsystem shall establish a session asynchronously when suitable network connectivity is available and when a request for a session is outstanding.
- The communications subsystem API shall be communications technology independent. The API does place requirements upon the underlying technology, which are presented in 8.2.

C.1.3 Functional requirements

- The link shall be established on demand by the Front End (FE) Application to the Back End (BE).
- A means shall be provided by which the BE can establish a connection to the FE Application.

NOTE The connection establishment time is undefined (e.g. the FE might be powered down).

- The link shall be organized on a “session” based paradigm. There will be distinct phases of activity relating to link setup, operation and teardown.
- The API shall provide positive indication to the FE Application of the current link state.
- Temporary outages of the link shall be transparent to the FE Application.
- When a communications session is lost it shall not be automatically re-established by the communications system.
- The API shall provide mechanisms to allow the BE to request specific information elements (as defined in ISO/TS 17575-1 and using the definitions therein) from the FE Application.
- The API shall provide mechanisms to deliver specific information elements (as defined in ISO/TS 17575-1 and using the definitions therein) to the BE.
- The abstract API shall be capable of carrying manufacturer-specific information which is outside of the scope of this part of ISO/TS 17575.
- The API shall provide positive indication to the FE Application that an ADU has definitely been delivered to the BE.
- All communications to and from the remote end shall be encrypted.

Annex D
(informative)

Examples of definitions for appropriate languages

D.1 General

The API has been deliberately designed to be language neutral, realizing that there are many different environments in which it will be used. In general the API should be implemented in a way that is compatible with the general use and idioms of the language under consideration, while retaining the general character and operation of the API.

D.2 API Definition in C

```
#ifndef _CEN17575_
#define _CEN17575_

#ifndef BOOL
#define BOOL char
#endif

#ifndef FALSE
#define FALSE 0
#define TRUE !FALSE
#endif

#ifndef BYTE
#define BYTE char
#endif

#ifndef WORD
#define WORD unsigned short int
#endif

// These macros allow us to reuse the Enums in string form for lookup
// Assumes that macros are indexed in order from 0
//
// For any enum AAA, these macros give you;
// typedef enum { x, y, z } AAAE
// const char *AAAS [] = {"x","y","z"}
// const char *toStringAAA( AAAE x )

#define MAKEstring(a) static const char *a ## S []={a}; const char *toString ##a(a ## E x) { return a
##S[x]; }
#define MAKEenum(a) typedef enum {a} a ## E; const char *toString ##a(a ## E x);
#define E(x) x

// Errors returned from the API
#define CEN175752Error                    \
    E(ERNoError),                         \
    E(ERNoInstance),                      \
    E(ERInterfacePreviouslyInitialised), \
    E(ERInsufficentPrivledge),            \
    E(ERSessionFailed),                  \
    E(ERInProgress),                     \
    E(ERUnknownStack),                   \
    E(ERNotActive),                      \
    E(ERInSession),                      \
    E(ERUnknownInstance),                \
    E(ERUnknownEndpoint),                \
    E(ERNoSession),                      \
    E(EREndinProgress),                  \
    E(ERHold),                           \
    E(ERDropInProgress),                 \
    E(ERMemoryError),                    \
    E(ERBadState),                       \
    E(ERNotSet),                         \
    E(ERUnknownError)

MAKEenum(CEN175752Error);

// Reasons for a callback or drop
#define CEN175752Reason  \
    E(REUnknown),        \
    E(RENormal),         \
    E(RECallback),       \
    E(RERemoteDrop),     \
    E(RETimed)

MAKEenum(CEN175752Reason);

// Responses to a session state query, or via a state change notification
#define  CEN175752State                     \
  E(STUnknownInstance),                     \
  E(STNoSession),                           \
  E(STStarting),                            \
  E(STSessionIdle),                         \
  E(STSendingElements),                     \
  E(STSendingElementsRequest),              \
  E(STSendingUnformattedMessage),           \
  E(STAwaitingElementsConfirm),             \
  E(STSessionRxElements),                   \
  E(STErrored),                             \
  E(STEnding)

MAKEenum(CEN175752State);

// Severity of a drop request
#define CEN175752Severity     \
    E(SENormal),              \
    E(SEUrgent),              \
    E(SEUnconditional)

MAKEenum(CEN175752Severity);

// This change in the definition of E is needed for the MAKEstring macro..which is used in the C file
#undef MAKEenum
#undef E
#define E(x) #x

// Definitions to simplify creation of up API
#define  cen175752APICALLBACKInstanceStateChange(x)  void  (x)(cen175752API  *instance,  CEN175752StateE
oldState, CEN175752StateE newState)
```

```
#define   cen175752APICALLBACKUnformattedADUReceived(x)   void   (x)(cen175752API   *instance,   WORD
unformattedMessageLen, BYTE *unformattedMessage)
#define cen175752APICALLBACKADURequest(x) void (x)(cen175752API *instance, void *elementReq)
#define cen175752APICALLBACKADUReceived(x) void (x)(cen175752API *instance, void *element)
#define cen175752APICALLBACKADUSent(x) void (x)(cen175752API *instance, BOOL sent)
#define cen175752APICALLBACKADUSend(x) void (x)(cen175752API *instance, BOOL canSend)
#define cen175752APICALLBACKSessionRequest(x) void (x)(char *handle);


typedef void cen175752API;

CEN175752ErrorE GetLastError(cen175752API *instance);

cen175752API *InitialiseInstance(char *stackID,
                  cen175752APICALLBACKInstanceStateChange(*InstanceStateChangeSet),
                  cen175752APICALLBACKUnformattedADUReceived(*UnformattedMessageReceivedSet),
                  cen175752APICALLBACKADURequest(*ElementRequestSet),
                  cen175752APICALLBACKADUReceived(*ElementReceivedSet),
                  cen175752APICALLBACKADUSent(*MessageSentSet),
                  cen175752APICALLBACKADUSend(*ElementSendSet),
                  cen175752APICALLBACKSessionRequest(*SessionRequestSet));

CEN175752ErrorE DropInstance(cen175752API *instance,
                  CEN175752SeverityE severity);

CEN175752ErrorE SetParameter(cen175752API *instance,
                      char *param,
                      char *val);

char *GetParameter(cen175752API *instance,
           char *param);

CEN175752ErrorE DeleteParameter(cen175752API *instance,
                         char *param);

CEN175752ErrorE StartSession(cen175752API *instance,
                      CEN175752ReasonE reason,
                      char *handle);

CEN175752ErrorE EndSession(cen175752API *instance,
                        CEN175752ReasonE reason);

CEN175752ErrorE SendUnformattedADU(cen175752API *instance,
                            WORD messageLen,
                            BYTE *message);

CEN175752ErrorE SendADUSetStart(cen175752API *instance);

WORD SendADU(cen175752API *instance,
         WORD messageLen,
         BYTE *message);

CEN175752ErrorE SendADUSetEnd(cen175752API *instance);

CEN175752StateE CommsQuery(cen175752API *instance);

BOOL StackAvail(cen175752API *instance);

#endif
```

Annex E
(informative)

Example of message flow

This annex contains an example of message flow for the case where the Front End Application wishes to establish a session, exchange some information with the Back End, and then terminate the session. This is illustrated in Figure E.1.

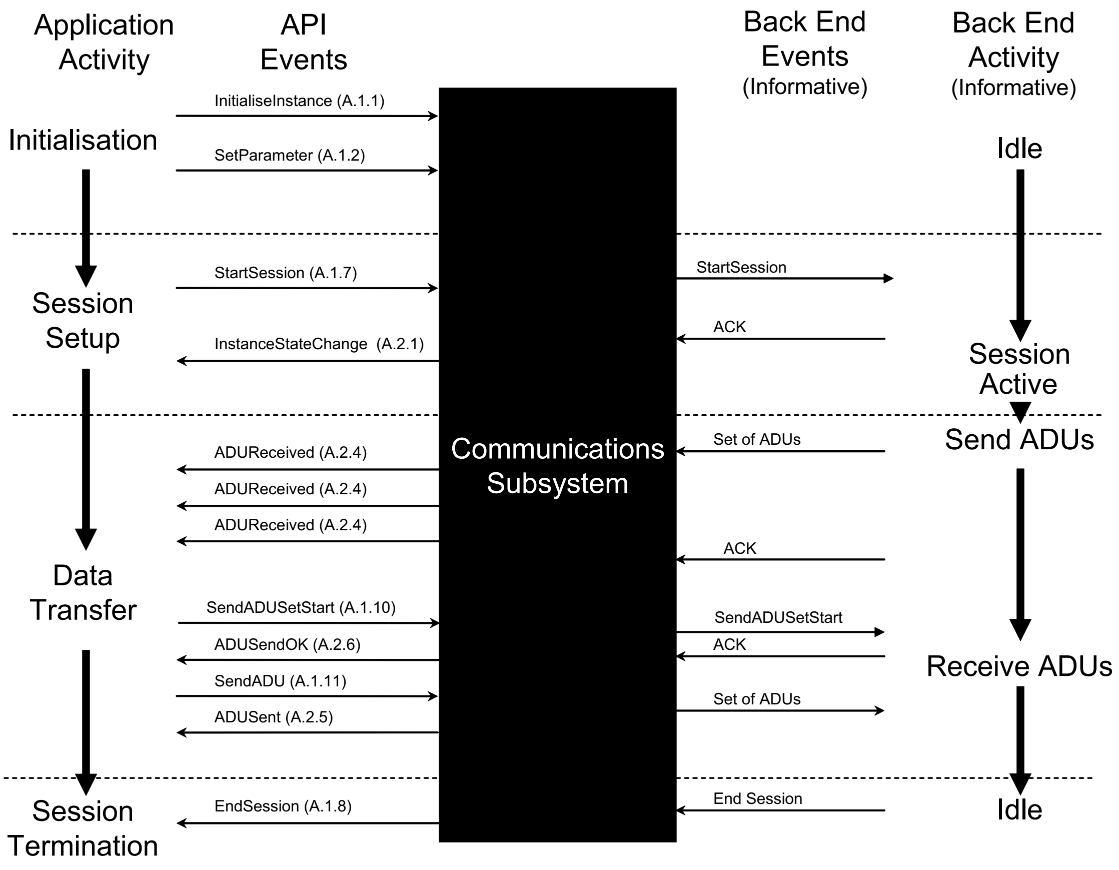

Figure E.1 — Example message flow: Front End initiates session

Bibliography

[1] ISO/IEC 9646-7, *Information technology — Open Systems Interconnection — Conformance testing methodology and framework — Part 7: Implementation Conformance Statements*

[2] ISO 17573:—[1], *Electronic fee collection — Systems architecture for vehicle related tolling*

[3] ISO/TS 17575-3:—[1], *Electronic fee collection — Application interface definition for autonomous systems — Part 3: Context data*

[4] ISO/TS 17575-4:—[1], *Electronic fee collection — Application interface definition for autonomous systems — Part 4: Roaming*

1) To be published.

ICS 03.220.20; 35.240.60

Price based on 25 pages