EUROPEAN STANDARD **EN 50221**

NORME EUROPÉENNE

EUROPÄISCHE NORM February 1997

UDC XXX

Descriptors: TBD

English Version

Common Interface Specification for Conditional Access and other Digital Video Broadcasting Decoder Applications

Foreword

This draft European Standard was prepared by the Technical Committee CENELEC TC 206, Broadcast Receiving Equipment.

The text of the draft was submitted to the Unique Acceptance Procedure and was approved by CENELEC as EN 50221 on 1997-02-15.

The following dates were fixed:

- latest date by which the EN has to be implemented at national level by publication of an identical national standard or by endorsement (dop) 1997-10-01

- latest date by which national standards conflicting with the EN have to be withdrawn (dow) 1997-10-01

Contents

1 Introduction and scope

A set of standards has been designed to be used in digital video broadcasting. These standards include source coding, channel coding, service information and decoder interfaces. In addition, a conditional access system is used when there is a need to control access to a broadcast service. It has been decided that the conditional access system need not be standardised, although a common scrambling algorithm is provided. It remains for broadcasters to access decoders with different conditional access systems and to ensure that they have choice of supply of such systems. A solution is to use the common scrambling algorithm and to execute solutions for access based on commercial agreements between operators. This solution can operate with single CA systems embedded in decoders.

A second solution is based on a standardised interface between a module and a host where CA and more generally defined proprietary functions may be implemented in the module. This solution also allows broadcasters to use modules containing solutions from different suppliers in the same broadcast system, thus increasing their choice and anti-piracy options. The scope of this document is to describe this common interface.

The decoder, referred to in this specification as the host, includes those functions that are necessary to receive MPEG-2 video, audio and data in the clear. This specification defines the interface between the host and the scrambling and CA applications, which will operate on an external module.

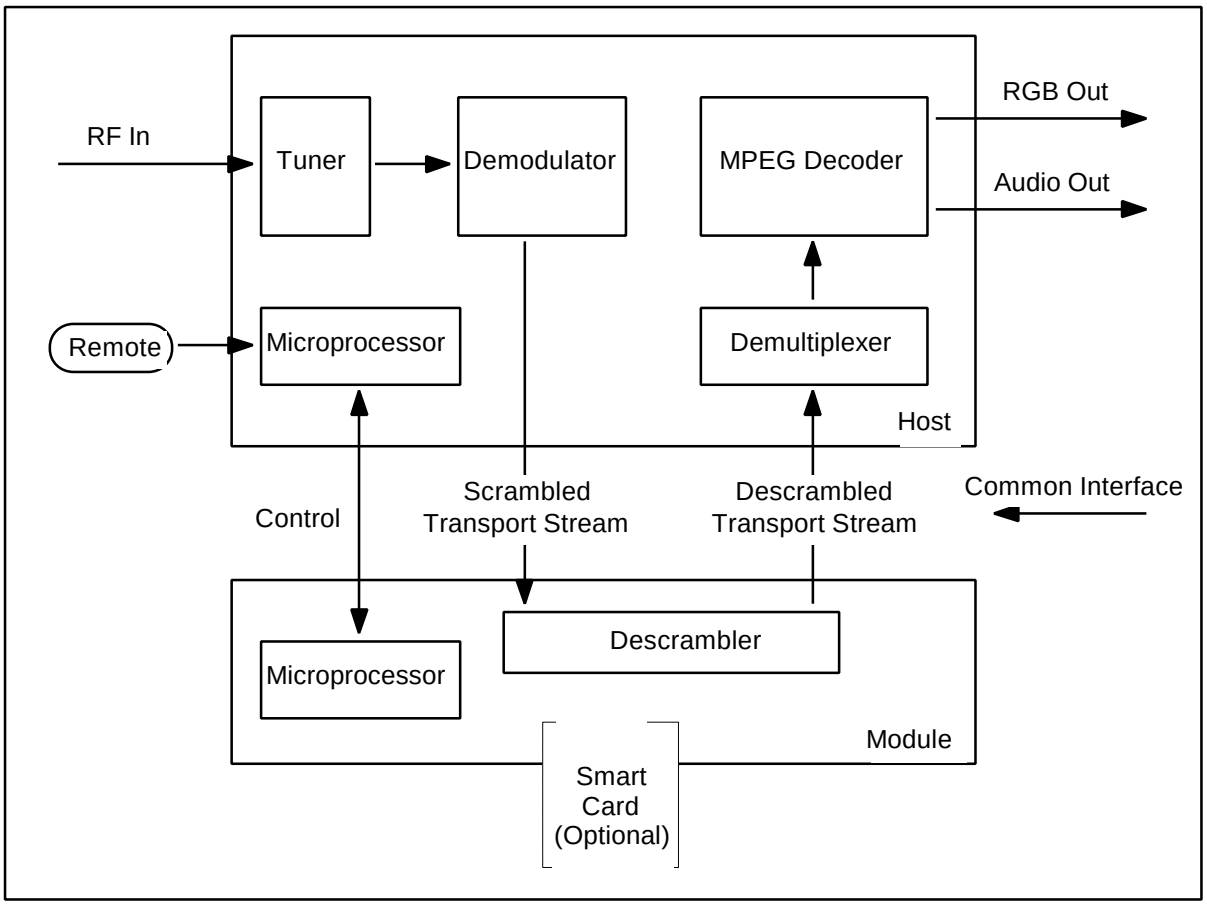

Figure 1: Example of single module in connection with host

Two logical interfaces, to be included on the same physical interface, are defined. The first interface is the MPEG-2 Transport Stream. The link and physical layers are defined in this specification and the higher layers are defined in the MPEG-2 specifications. The second interface, the command interface, carries commands between the host and the module. Six layers are defined for this interface. An example of a single module in connection with a host is shown in figure 1.

This specification only defines those aspects of the host that are required to completely specify the interactions across the interface. The specification assumes nothing about the host design except to define a set of services which are required of the host in order to allow the module to operate.

The specification does not define the operation or functionality of a conditional access system application on the module. The applications which may be performed by a module communicating across the interface are not limited to conditional access or to those described in this specification. More than one module may be supported concurrently.

2 Definitions

For the purposes of this standard, the following definitons apply:

application : An application runs in a module, communicating with the host, and provides facilities to the user over and above those provided directly by the host. An application may process the Transport Stream.

host : A device where module(s) can be connected, for example : an IRD, a VCR, a PC ...

module : A small device, not working by itself, designed to run specialised tasks in association with a host, for example : a conditional access sub system, an electronic program guide application module, or to provide resources required by an application but not provided directly by the host

resource : A unit of functionality provided by the host for use by a module. A resource defines a set of objects exchanged between module and host by which the module uses the resource.

service : A set of elementary streams offered to the user as a program. They are related by a common synchronisation. They are made of different data, i.e., video, audio, subtitles, other data...

transport stream : MPEG-2 Transport Stream.

3 Normative references

This European Standard incorporates by dated or undated reference, provisions from other publications. These normative references are cited at the appropriate places in the text and the publications are listed hereafter. For dated references, subsequent amendments to or revisions of any of these publications apply to this European Standard only when incorporated in it by amendment or revision. For undated references the latest edition of the publication referred to applies (including amendments).

[1]	ISO/IEC 13818-1	Information technology - Generic coding of moving pictures and associated audio information: Systems
[2]	ISO 8824 1987	Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)
[3]	ISO 8825 1987	Open Systems Interconnection - Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1)
[4]	ETS 300 468	Specification for Service Information (SI) in Digital Video Broadcasting (DVB) systems
[5]	ETR 162	Allocation of Service Information (SI) codes for Digital Video Broadcasting (DVB) Systems
[6]	PC Card Standard	Volume 2 - Electrical Specification, February 1995, Personal Computer Memory Card International Association, Sunnyvale, California
[7]	PC Card Standard	Volume 3 - Physical Specification, February 1995, Personal Computer Memory Card International Association, Sunnyvale, California
[8]	PC Card Standard	Volume 4 - Metaformat Specification, February 1995, Personal Computer Memory Card International Association, Sunnyvale, California
[9]	prETS 300 743	DVB Subtitling Specification

4 Design philosophy

4.1 Layering

The specification is described in layers in order to accommodate future variations in implementation. The application and session layers are defined for all applications of the common interface. The transport and link layers may be dependent on the physical layer used in a particular implementation. The physical interface is defined within this specification and includes the complete physical specification of the module

The layering of the specification allows flexibility in the use of the interface for a range of applications beyond CA. It also allows for multiple instance of CA processes to exist for the same host.

A representation of the basic layering on the command interface is shown in figure 2. The host may set up transport connections with more than one module, which may be connected directly or indirectly to the host. Each connection is maintained while the module is present. Each module may manage a number of different sessions with the host.

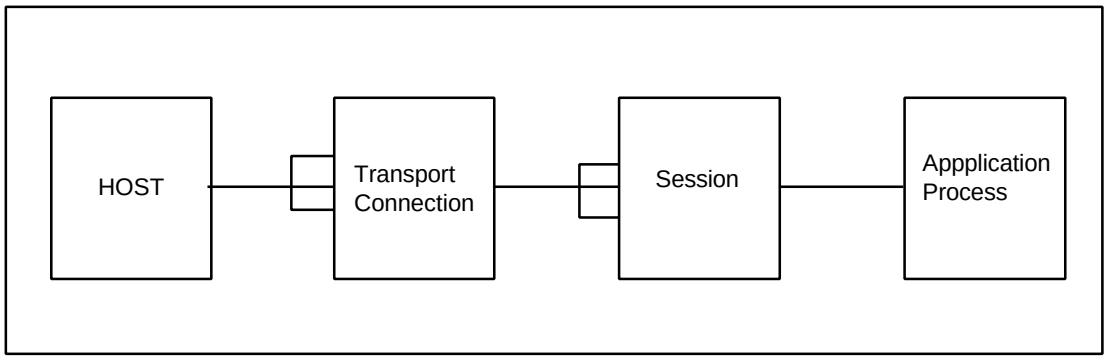

Figure 2: Layering on the command interface

4.2 Physical implementation

The baseline specification includes the implementation on a physical interface compatible with the PC Card standard used in the Personal Computer industry. Other physical implementations are allowed for in the future.

4.3 Client-server

The interface is designed on the principle that applications, as clients, use resources provided by a server. The applications reside on a module and resources can be served either by the host or another module in a way managed by the host. The term ‘resources’ has been used in preference to ‘services’ as that term is common in the broadcasting field for TV and radio services and there is a need to avoid confusion.

4.4 Coding of data

The communication of data across the command interface is defined in terms of objects. The objects are coded by means of a general Tag-Length-Value coding derived from that used to code ASN.1 syntax (see [2] and [3]). This is generally extensible. There is a particular transport layer coding for the PC Card implementation but it may be different in other physical implementations. However the semantics would be identical.

4.5 Extensibility

The higher layers have been designed to be extensible. As indicated above, the TLV coding used is extensible so that new objects can be added, and existing objects can be extended. There is no problem about running out of tag coding space, or length restrictions on the values. The Resource Manager resource provides a mechanism for extending the range of resources provided by hosts, both for CA purposes and for other module-based applications.

4.6 Incorporation of existing standards

Existing standards have been used, where possible and appropriate, as building blocks for this specification. This gives important time-to-market benefits, as all the standards development work has already been done. It also gives implementation benefits in that software and hardware already developed for existing standards may be re-used here, with potential cost benefits.

5 Description and architecture

5.1 Overview

A partial logical architecture has been assumed for a host in order to define the place in the host where the common interface can logically occur. The impact upon the freedom of choice for host designers in other respects has been minimised. Figure 1 shows a simplified picture of a typical host architecture and the positioning of the interface within it. Note that there can be more than one instance of the interface on a host.

The common interface consists of two components, the Transport Stream Interface and the Command Interface. Both are layered to make the overall interface design and implementation easier. The upper layers are common to all implementations but alternative lower-layer implementations are possible. This specification includes one based upon the PC Card standard but others may be included in future versions.

5.2 Transport Stream Interface

The Transport Stream Interface carries MPEG-2 transport packets in both directions. If the module gives access to any services in the transport stream and those services have been selected by the host, then the packets carrying those services will be returned descrambled, and the other packets are not modified. On the Transport Stream Interface a constant delay through the module and any associated physical layer conditioning logic is preserved under most conditions (see 5.4.2). The Transport Stream Interface layers are shown in figure 3 below. The Transport Layer and all upper layers are defined in the MPEG-2 specification - ISO 13818.

Upper Layers
Transport Layer
PC Card Link Layer
PC Card Physical Layer

Figure 3: Transport Stream Interface Layers

5.3 Command Interface

The Command Interface carries all the communication between the application(s) running in the module and the host. The communication protocols on this interface are defined in several layers in order to provide the necessary functionality. This functionality includes: the ability to support multiple modules on one host, the ability to support complex combinations of transaction between module and host, and an extensible set of functional primitives (objects) which allow the host to provide resources to the module. The layering is shown in figure 4 below.

The PC Card implementation described in this specification has its own Physical and Link layers, and also its own Transport lower sublayer. A future different physical implementation is likely to differ in these layers and any difference will be restricted to these layers. The implementation-specific features of the Transport lower sublayer are limited to coding and specific details of the message exchange protocol, and the common upper sublayer defines identification, initiation and termination of Transport layer connections. The Session, Resource and Application layers are common to all physical implementations.

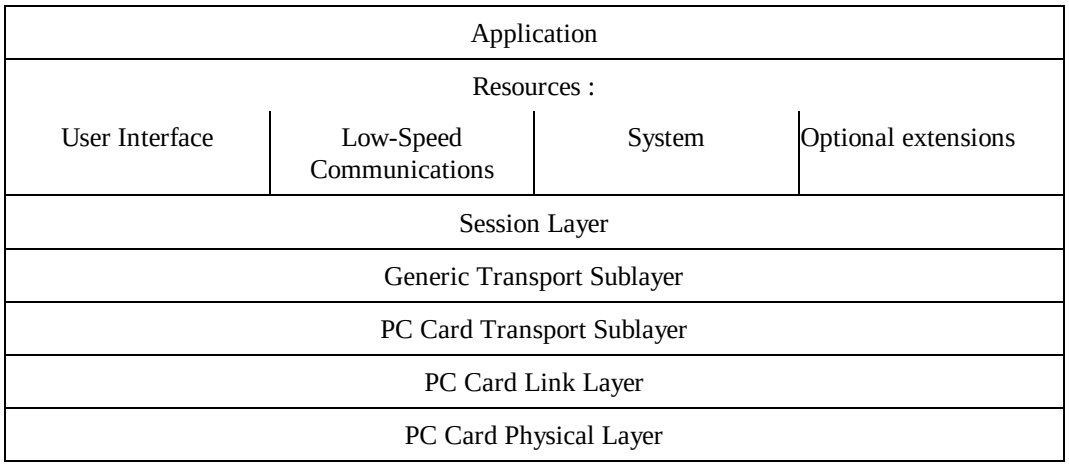

Figure 4: Command Interface Layers

As far as possible the Application layer of the interface has been designed to be free of specific application semantics. Communication is in terms of resources, such as User Interface interaction, and low-speed communications, that the host provides to the application(s) running on a module. This strategy makes it very much easier to provide modules performing other tasks than just Conditional Access.

5.4 Physical requirements

5.4.1 Introduction

This clause defines the requirements the Physical Layer must meet in order to carry out all the required functions. The following Physical Layer characteristics are not constrained here, although the specification for any Physical Layer used will define them: mechanical and electrical connection between the host and the module, i.e. socket type & size, number of pins, voltages, impedances, power limits.

Requirements and limits on the following Physical Layer characteristics are defined here:

- Transport Stream and Command logical connections;
- data rates;
- connection & disconnection behaviour;
- low-level initialisation;
- use of multiple modules.

5.4.2 Data and Command logical connections

The Physical Layer shall support independent both-way logical connections for the Transport Stream and for commands.

The Transport Stream Interface shall accept an MPEG-2 Transport Stream, consisting of a sequence of Transport Packets, either contiguously or separated by null data. The returned Transport Stream may have some of the incoming transport packets returned in a descrambled form. The Transport Stream Interface is subject to the following restrictions:

1 When the module is the source of a transport stream its output shall comply with ISO/IEC 13818-9.

2 Each output packet shall be contiguous if the module is the source of the packet or the input packet is contiguous.

3 A module shall introduce a constant delay when processing an input transport packet, with a maximum delay variation (tmdv) applied to any byte given by the following formula:

$tmdv_{max}$ = (n * TMCLKI) + (2 * TMCLKO).

and

$tmdv_{max}$ <= 1 microsecond when n = 0

where:

tmdv	=	Module Delay Variation
n	=	Number of gaps present within the corresponding input transport packet
TMCLKI	=	Input data clock period
TMCLKO	=	Output data clock period

* A `gap' is defined to be one MCLKI rising edge for which the MIVAL signal is inactive.
* All hosts are strongly recommended to output contiguous transport packets.
* Hosts may only output non-contiguous transport packets if they implement less than 3 common interface sockets.
* Inter packet gaps may vary considerably.

4 A CI compliant host should be designed to support Nm modules. Nm is the greater of the number of CI sockets implemented by the host or 16. It should tolerate the jitter resulting from Nm modules plus the jitter in the input transport stream. The worst case jitter may arise either from the host's own input followed by Nm modules or an input module with a ISO/IEC 13818-9 compliant output followed by (Nm - 1) modules.

5 All interfaces shall support a data rate of at least 58 Mb/s averaged over the period between the sync bytes of successive transport packets.

6 All interfaces shall support a minimum byte transfer clock period of 111 ns.

The Command Interface shall transfer commands as defined by the appropriate Transport Layer part of this specification in both directions. The data rate supported in each direction shall be at least 3,5 Megabits/sec.

5.4.3 Connection and disconnection behaviour

The Physical layer shall support connection and disconnection of the module at any time, whether the host is powered or not. Connection or disconnection shall not cause any electrical damage to either module or host, and shall not cause any spurious modification of stored non-volatile data in the module. When a module is not connected the Transport Stream Interface shall bypass the module, and the Command Interface to that module shall be inactive. On connection of a module, the host shall initiate a low-level initialisation sequence with the module. This will carry out whatever low-level connection establishment procedures are used by the particular Physical Layer, and then establish that the module is a conformant DVB module. If successfully completed, the host shall establish the Transport Stream connection by inserting the module into the host's Transport Stream path. It is acceptable that some Transport Stream data is lost during this process. At the same time a Transport Layer connection shall be established on the Command Interface to allow Application Layer initialisation to take place and normal Application Layer communications to proceed.

If the Physical Layer is used in other applications than as a DVB-conformant module connection, and if a non-conformant module is connected to the host, no damage shall be caused to the module or the host, and the host shall not attempt to complete initialisation as though it were a DVB-conformant module. Optionally, the host may signal to the user that an unrecognised module has been connected.

On disconnection of the module, the host shall remove the module from the Transport Stream data path. It is acceptable that some Transport Stream data is lost during this process. Also, the Command Interface connection shall be terminated by the host.

5.4.4 Multiple modules

The Application Layer places no limit on the number of modules which may be connected to the host at any time. However, particular Physical Layers and particular host design choices may do so. The Physical Layer specification must allow there to be several modules connected simultaneously to the host, even though a minimum host design may only provide for one connection. Ideally the Physical Layer specification should place no hard limit on the number of modules, but if a limit is imposed, then it shall be set at no less than 15 modules.

Where there is provision for more than one module to be connected, the Transport Stream Interface connection shall be daisy-chained through each module in turn, as illustrated in figure 5 below. The host shall maintain separate and simultaneous Command Interface connections to each module, so that transactions between host and module are treated independently for each module. When a module is unplugged the Command Interface transport layer connection to any other module shall not be disturbed or terminated.

When several modules are connected to a host, the host should be able to select the module(s) relevant for the descrambling of the selected service(s).

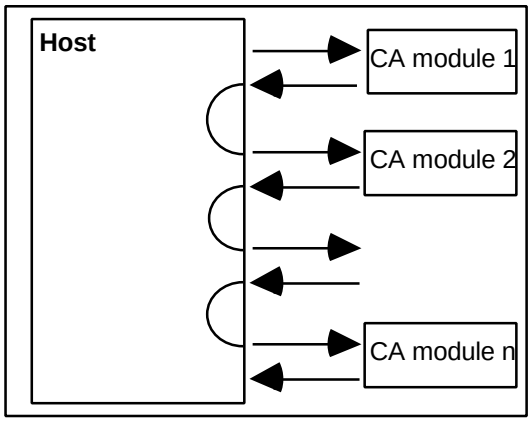

Figure 5: Transport Stream Interface chaining between modules

5.5 Operational example

To illustrate some of the features described above consider this example of the processes that occur when a PC Card module is plugged into a host. The PC Card initialisation commences with sensing of the module being plugged in by sense pins on the interface. The host then reads the Card information Structure residing in the Attribute Memory of the module. This contains low-level configuration information for the module, such as PC Card read and write addresses used by the module, and indicates to the host that it is a DVB-conformant module. The host now turns off the Transport Stream Interface bypass link and allows the transport packets to flow through the module. This introduces a delay, and consequently a short gap in the Transport Stream data, but this is unavoidable. At the same time the physical layer interface initialisation process takes place to negotiate the buffer size to be used for communication. At this point the physical layer initialisation process is complete and the upper-layer initialisation process, common to all physical implementations, commences with the host creating a Transport Layer connection to the module. This process and the rest of the upper-layer initialisation process are described elsewhere in this document.

The initialisation process will be logically similar for other physical implementations though the details will differ.

6 Transport Stream Interface (TSI)

6.1 TSI - physical, link layers

These layers depend on the physical implementation of the module.

6.2 TSI - transport layer

The transport layer used is the same as the MPEG-2 System transport layer. Data travelling over the transport stream interface is organised in MPEG-2 Transport Packets. The whole MPEG-2 multiplex is sent over this transport stream interface and is received back fully or partly descrambled. If the packet is not scrambled, the module returns it as is. If it is scrambled and the packet belongs to the selected service and the module can give access to that service, then the module returns the corresponding descrambled packet with the transport_scrambling_control flag set to '00'.

If scrambling is performed at Packetised Elementary Stream (PES) level, then the module reacts in the same way and under the same conditions as above, and returns the corresponding descrambled PES with the PES_scrambling_control flag set to '00'.

The transport packet and the PES packet are completely defined in the MPEG-2 System specification [1].

6.3 TSI - upper layers

Apart from the Packetised Elementary Stream, any layering or structure of the MPEG-2 data above the Transport Stream layer is not relevant to this specification. However the specification does assume that the module will find and extract certain data required for its operation, such as ECM and EMM messages, directly from the Transport Stream.

7 Command Interface - Transport & Session Layers

The communication of data across the command interface is defined in terms of objects. The objects are coded by means of a general Tag-Length-Value coding derived from that used to code ASN.1 syntax.

Table 1: Length field used by all Protocol Data Units at Transport, Session & Application Layers

Syntax	No. of bits	Mnemonic
length_field() {		
size_indicator	**1**	**bslbf**
if (size_indicator == 0)		
length_value	**7**	**uimsbf**
else if (size_indicator == 1) {		
length_field_size	**7**	**uimsbf**
for (i=0; i<length_field_size; i++) {		
length_value_byte	**8**	**bslbf**
}		
}		
}		

This clause describes the ASN.1 objects for the Transport and Session Layers that travel over the command interface. For all these objects, and for the Application Layer objects in clause 8, the coding in table 1 applies for the Length field, which indicates the number of bytes in the following Value field.

Size_indicator is the first bit of the length_field. If size_indicator = 0, the length of the data field is coded in the succeeding 7 bits. Any length from 0 to 127 can thus be encoded on one byte. If the length exceeds 127, then size_indicator is set to 1. In this case, the succeeding 7 bits code the number of subsequent bytes in the length field. Those subsequent bytes shall be concatenated, first byte at the most significant end, to encode an integer value. Any value field length up to 65535 can thus be encoded by three bytes.

The indefinite length format specified by the basic encoding rules of ASN.1 (see [3]) is not used.

7.1 Generic Transport Layer

7.1.1 Introduction

The Transport Layer of the Command Interface operates on top of a Link Layer provided by the particular physical implementation used. For the baseline PC Card physical implementation the Link Layer is described in annex A. The transport protocol assumes that the Link Layer is reliable, that is, data is conveyed in the correct order and with no deletion or repetition of data.

The transport protocol is a command-response protocol where the host sends a command to the module, using a Command Transport Protocol Data Unit (C_TPDU) and waits for a response from the module with a Response Transport Protocol Data Unit (R_TPDU). The module cannot initiate communication: it must wait for the host to poll it or send it data first. The protocol is supported by eleven Transport Layer objects. Some of them appear only in C_TPDUs from the host, some only in R_TPDUs from the module and some can appear in either. Create_T_C and C_T_C_Reply, create new Transport Connections. Delete_T_C and D_T_C_Reply, clear them down. Request_T_C and New_T_C allow a module to request the host to create a new Transport Connection. T_C_Error allows error conditions to be signalled. T_SB carries status information from module to host. T_RCV requests waiting data from a module and T_Data_More and T_Data_Last convey data from higher layers between host and module. T_Data_Last with an empty data field is used by the host to poll regularly for data from the module when it has nothing to send itself.

A C_TPDU from the host contains only one Transport Protocol Object. A R_TPDU from a module may carry one or two Transport Protocol Objects. The sole object or second object of a pair in a R_TPDU is always a T_SB object.

7.1.2 Transport protocol objects

All transport layer objects contain a transport connection identifier. This is one octet, allowing up to 255 Transport Layer connections to be active on the host simultaneously. Transport connection identifier value 0 is reserved. The identifier value is always assigned by the host. The protocol is described in detail here as it is common to all physical implementations but the objects are only described in general terms. The detailed coding of the objects depends upon the particular physical layer used. The coding for the PC Card physical implementation is described in annex A.

The host shall allow at least 16 transport connections to be created per module socket supported but preferably all 255 connections distributed amongst the module sockets.

1 Create_T_C creates the Transport Connection. It is only issued by the host and carries the transport connection identifier value for the connection to be established.

2 C_T_C_Reply is the response from the target module to Create_T_C and carries the transport connection identifier for the created connection.

3 Delete_T_C deletes an existing Transport Connection. It has as a parameter the transport connection identifier for the connection to be deleted. It can be issued by either host or module. If issued by the module it does so in response to a poll or data from the host.

4 D_T_C_Reply is the reply to the delete. In some circumstances this reply may not reach its destination, so the Delete_T_C object has a time-out associated with it. If the time-out matures before the reply is received then all actions which would have been taken on receipt of the reply can be taken at the time-out.

5 Request_T_C requests the host to create a new Transport Connection. It is sent on an existing Transport Connection from that module. It is sent in response to a poll or data from the host.

6 New_T_C is the response to Request_T_C. It is sent on the same Transport Connection as the Request_T_C object, and carries the transport connection identifier of the new connection. New_T_C is immediately followed by a Create_T_C object for the new connection, which sets up the Transport Connection proper.

7 T_C_Error is sent to signal an error condition and carries a 1-byte error code specifying the error. In this version this is only sent in response to Request_T_C to signal that no more Transport Connections are available.

8 T_SB is sent as a reply to all objects from the host, either appended to other protocol objects or sent on its own, as appropriate. It carries one byte which indicates if the module has data available to send.

9 T_RCV is sent by the host to request that data the module wishes to send (signalled in a previous T_SB from the module) be returned to the host.

10 T_Data_More and T_Data_Last convey data between host and module, and can be in either a C_TPDU or a R_TPDU. From the module they are only ever sent in response to an explicit request by a T_RCV from the host. T_Data_More is used if a Protocol Data Unit (PDU) from a higher layer has to be split into fragments for sending due to external constraints on the size of data transfers. It indicates that at least one more fragment of the upper-layer PDU will be sent after this one. T_Data_Last indicates the last or only fragment of an upper-layer PDU.

7.1.3 Transport protocol

Figures 6 and 7 show the state transition diagrams for connection set-up and clear-down on the host side and the module side respectively. Each state transition arc is labelled with the event that causes that transition. If the transition also causes an object to be sent, then this is indicated with boxed text.

When the host wishes to set up a transport connection to a module, it sends the Create_T_C object and moves to state 'In Creation'. The module shall reply directly with a C_T_C_Reply object. If after a time-out period the module does not respond, then the host returns to the idle state (via the 'Timeout' arc). The host will not transmit or poll again on that particular transport connection, and a late C_T_C_Reply will be ignored. If, subsequently, the host re-uses the same transport connection identifier, then the module will receive Create_T_C again, and from this it shall infer that the old transport connection is dead, and a new one is being set up.

When the module replies with C_T_C_Reply the host moves to the 'Active' state of the connection. If the host has data to send, it can now do so, but otherwise it issues a poll and then polls regularly thereafter on the connection.

If the host wishes to terminate the transport connection, it sends a Delete_T_C object and moves to the 'In Deletion' state. It then returns to the 'Idle' state upon receipt of a D_T_C_Reply object, or after a time-out if none is received. If the host receives a Delete_T_C object from the module it issues a D_T_C_Reply object and goes directly to the idle state. Except for the 'Active' state, any object received in any state which is not expected is ignored.

In the 'Active' state the host issues polls periodically, or sends data if it has an upper-layer PDU to send. In response it receives a T_SB object, preceded by a Request_T_C or Delete_T_C object if that is what the module wants to do.

In the ‘Active’ state, data can be sent by the host at any time. If the module wishes to send data it must wait for a message from the host - normally data or a poll - and then indicate that it has data available in the T_SB reply. The host will then at some point - not necessarily immediately - send a T_RCV request to the module to which the module responds by sending the waiting data in a T_Data object. Where T_Data_More is used, each subsequent fragment must wait for another T_RCV from the host before it can be sent.

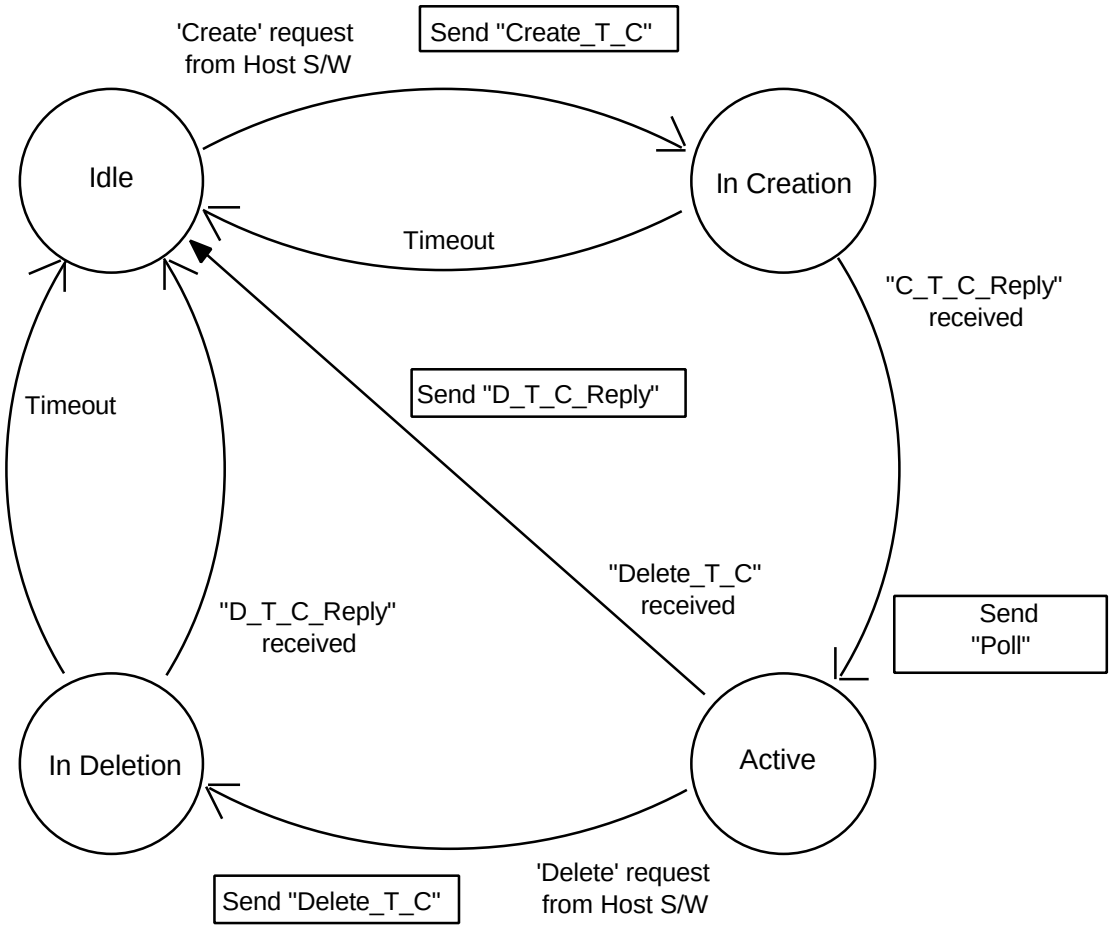

Figure 6: State transition diagram for the host side of the transport protocol

Table 2: Objects expected to be received in the states of a transport connection on the host

State	**Expected objects - Host**
Idle	None
In Creation	C_T_C_Reply (+ T_SB)
Active	T_Data_More, T_Data_Last, Request_T_C, Delete_T_C, T_SB
In Deletion	D_T_C_Reply (+ T_SB)

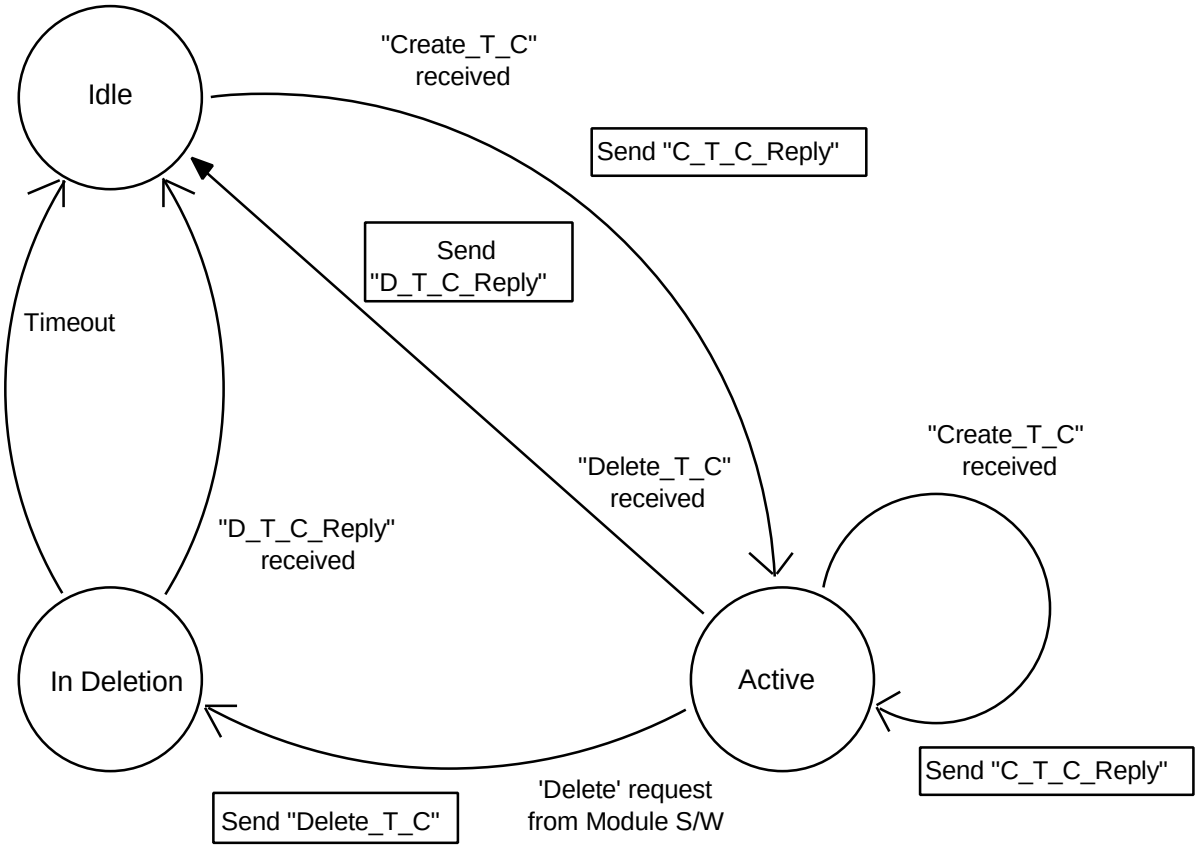

Figure 7: State transition diagram for the module side of the transport protocol

Table 3 - Objects expected to be received in the states of a transport connection on the module

State	**Expected objects - Module**
Idle	Create_T_C
Active	Create_T_C, T_Data_More, T_Data_Last, New_T_C, Delete_T_C, T_RCV, T_C_Error
In Deletion	D_T_C_Reply

If a module wishes to set up another Transport Connection it shall send a Request_T_C object either in response to a poll or data. If it can meet the request the host will reply with a New_T_C containing the transport connection identifier for the new connection, immediately followed by a Create_T_C object to create the connection. If the host cannot meet the request because all transport connection identifiers are in use then it shall reply with a T_C_Error containing the appropriate error code.

An example of an object transfer sequence to create and use a Transport Connection is illustrated in figure 8 below.

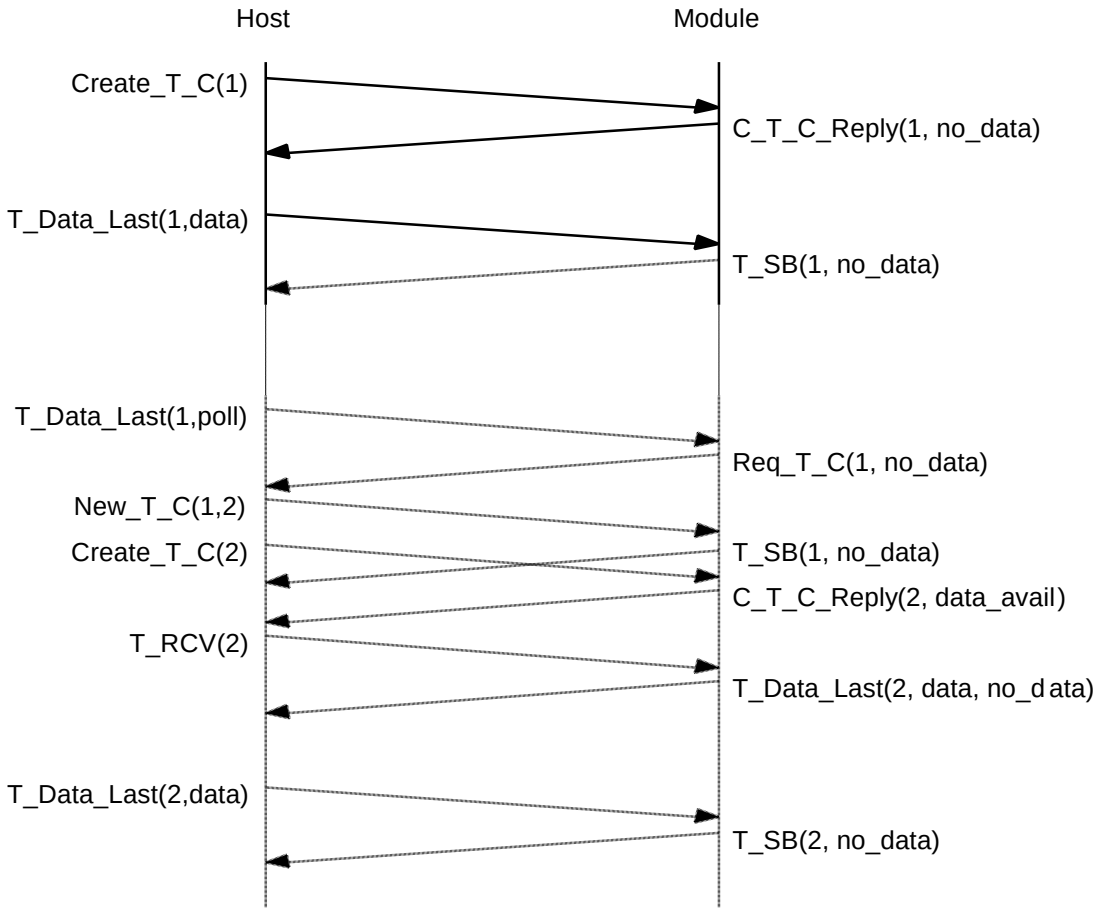

Figure 8:Object transfer sequence example for the tansport potocol

In this example let us assume that the module has just been plugged in and a physical connection has been established (PC Card initialisation, etc.). The host now issues a Create_T_C for Transport Connection number 1 (if there were already other modules plugged in then this would be a higher number). The module replies immediately with C_T_C_Reply for Transport Connection 1, also indicating it has no data to send. The host now sends some data with a T_Data_Last and the module responds with just a T_SB indicating no data to send. Some time later the host polls the module with an empty T_Data_Last, and the module responds with a Request_T_C saying it wishes to have a new Transport Connection created, and also indicating (in the appended T_SB) that it has no data to send on connection 1. The host replies with New_T_C indicating that Transport Connection 2 will be set up. This is immediately followed by Create_T_C for Transport Connection 2. The module responds with a T_SB to the first and a C_T_C_Reply to the second, also indicating that it has data to send on this connection. The host responds with T_RCV to receive the data, and the module responds with T_Data_Last containing the data. The host replies with data of its own and the module responds to that indicating it has no further data to send. Both Transport Connections now persist until either module or host deletes one, and the host polls periodically on both connections with an empty T_Data_Last.

7.2 Session Layer

7.2.1 Introduction

The Session Layer provides the mechanism by which applications communicate with and make use of resources. The resource is a mechanism for encapsulating functionality at the Application Layer and is described fully in 8.2.

Resources vary in the number of simultaneous sessions they can support. Some resources support only one. If a second application tries to request a session to such a resource already in use then it will receive a 'resource busy' reply. Other resources can support more than one simultaneous session, in which case resource requests will be honoured up to some limit defined by the resource. An example of the latter would be the display resource, which in some host implementations may be able to support simultaneous displays in different windows.

7.2.2 Session protocol objects

The session objects are described in general terms here, as is the protocol, but the detailed coding of the objects is described in later subclauses.

- open_session_request is issued by an application over its transport connection to the host requesting the use of a resource. The host may support the resource directly, or it may be supported by another module (over a second transport connection), in which case the host uses the create_session object to extend the session over the appropriate transport connection.

- open_session_response is returned by the host to an application that requested a resource in order to allocate a session number or to tell the module that its request could not be fulfilled.

- create_session is issued by the host to a resource provider in a module in order to extend a session request from an application on another transport connection.

- create_session_response is returned by the resource provider in a module to the host so the host can tell the originating module whether the session could be opened..

- close_session_request is issued by a module or by the host to close a session.

- close_session_response is issued by a module or by the host to acknowledge the closing of the session.

- session_number always precedes the body of the SPDU containing APDU(s).

7.2.3 Session protocol

The dialogue in the session layer is initiated by a module or by the host. Two examples are illustrated below. In the first example (figure 9), a module A wishes to use a resource which is provided by the host. In the second example (figure 10), a module A requests to use a resource provided by module B.

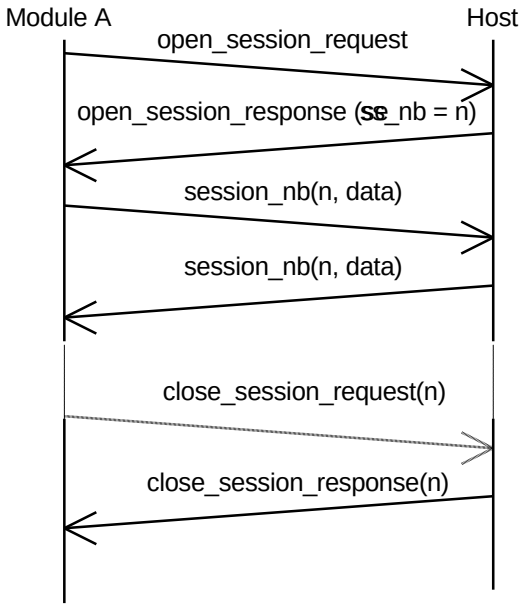

Figure 9: Session to a host-provided resource

Module A requests a session to be opened to a resource on its transport connection. Since the host provides the resource itself it replies directly with a session number in its open session response. Communication now proceeds with application layer data preceded by session_number objects. Eventually the session is closed, in this example, by module A, but it could also have been closed by the host, for example if the resource became unavailable for any reason.

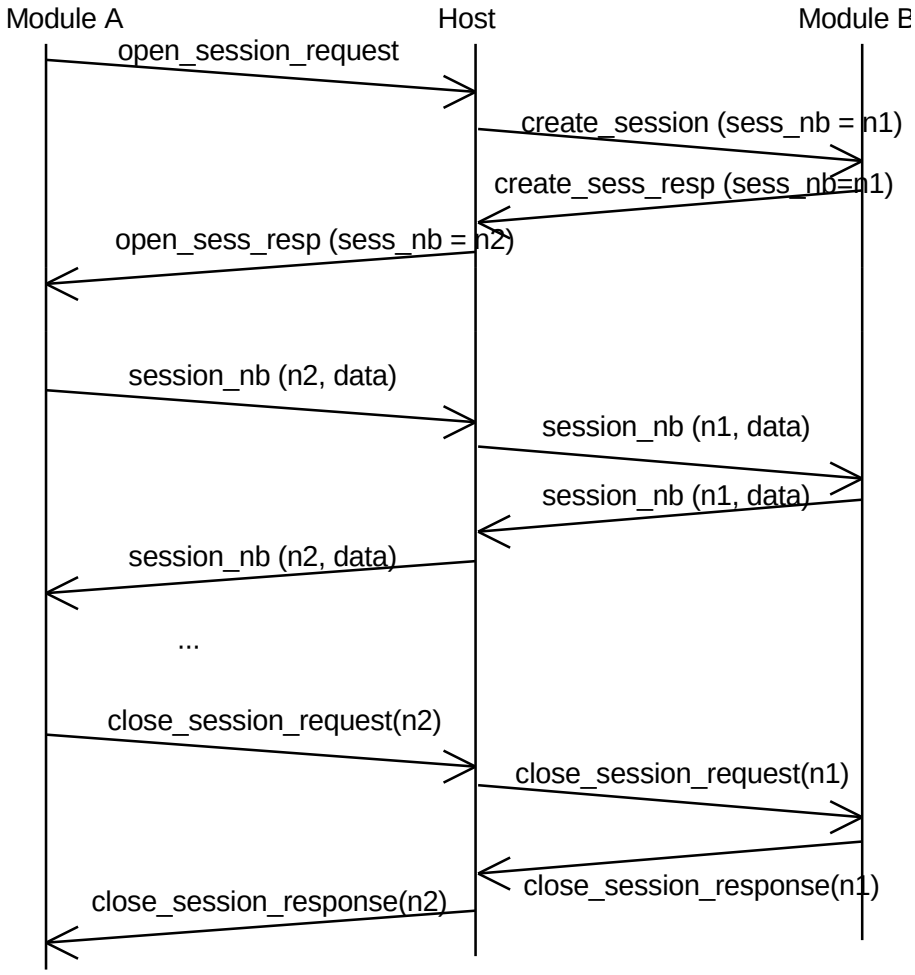

Figure 10: Session to a module-provided resource

Module A requests a session to be opened to a resource on its transport connection. The host knows that this resource is provided by module B, so the host extends the session on the resource provider's transport connection with a create_session object. The resource provider responds to this and the host in turn responds to the original requester. Communication of application layer data now proceeds, and the host performs the necessary routing between incoming and outgoing transport connections. Finally the session is closed, in this example by module A, but it could also have been closed by the host or by module B. The session numbers n1 and n2 are allocated by the host. In this example they are shown as being different. It is a host implementation choice whether it uses the same or different session numbers on each transport connection. The Application Layer can make no assumption about whether the session numbers are the same or not at each end of the session.

7.2.4 SPDU structure

The session layer uses a Session Protocol Data Unit (SPDU) structure to exchange data at session level either from the host to the module or from the module to the host.

The SPDU structure is illustrated by figure 11 and table 4 below :

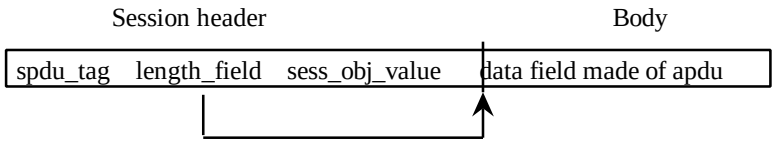

Figure 11: SPDU structure

Table 4: SPDU Coding

Syntax	No. of bits	Mnemonic
SPDU() {		
spdu_tag	**8**	**uimsbf**
length_field()		
for (i=0;i<length_value;i++) {		
session_object_value byte	**8**	**uimsbf**
}		
for (i=0;i<N;i++) {		
apdu()		
}		
}		

The SPDU is made of two parts :

- a **mandatory** session header made of a Tag value spdu_tag, a length_field coding the length of the session object value field and a session object value. The session objects are described in 7.2.6. below. Note that the length field does *not* include the length of any following APDUs.
- a conditional body of variable length which contains an integer number of APDUs belonging to the same session (see application layer). The presence of the body depends on the session header.

7.2.5 Transportation of SPDU

A SPDU is transported in the data field of one or several TPDU. See TPDU description of each physical module implementation for more information.

7.2.6 Session headers description

The objects listed below are session header objects. Only one SPDU header is followed by a data field - the session_number object - which is always followed by a SPDU body containing one or several APDUs.

7.2.6.1 Open Session Request

This object is issued by the module to the host in order to request the opening of a session between the module and one resource provided either by the host or by a module. The resource_identifier must match in both class and type a resource that the host has in its list of available resources. If the version field of the supplied resource identifier is zero, then the host will use the current version in its list. If the version number in the request is less than or equal to the current version number in the host's list then the current version is used. If the requested version number is higher than the version in the host's list, then the host will refuse the request with the appropriate return code.

Table 5: Open Session Request coding

Syntax	No. of bits	Mnemonic
open_session_request () {		
open_session_request_tag	**8**	**uimsbf**
length_field() = 4		
resource_identifier() /* see 8.2.2. */		
}		

7.2.6.2 Open Session Response

This object is issued by the host to the module in order to allocate a session number or to tell the module that its request could not be met.

Table 6: Open Session Response coding

Syntax	No. of bits	Mnemonic
open_session_response () {		
open_session_response_tag	**8**	**uimsbf**
length_field() = 7		
session_status	**8**	**uimsbf**
resource_identifier()		
session_nb	**16**	**uimsbf**
}		

session_status values

Table 7: Open Session Status values

session status	session_status value (hex)
session is opened	00
session not opened, resource non-existent	F0
session not opened, resource exists but unavailable	F1
session not opened, resource exists but version lower than requested	F2
session not opened, resource busy	F3
other	reserved

resource_identifier
The host returns the actual resource_identifier of the resource requested, with the current version number. If the response is ‘resource non-existent’ then the resource_identifier field will be identical to that supplied in the open request.

session_nb
Number allocated by the host for the requested session. Value 0 is reserved. The session_nb will be used for all subsequent exchanges of APDUs between the module and the resource (through the host) until the session is closed. When the session could not be opened (session_status not equal to 0), the session_nb value has no meaning.

7.2.6.3 Create Session

This object is issued by the host to a module providing a resource in order to request the opening of a session.

Table 8: Create Session coding

Syntax	No. of bits	Mnemonic
create_session () {		
create_session_tag	**8**	**uimsbf**
length_field() = 6		
resource_identifier()		
session_nb	**16**	**uimsbf**
}		

resource_identifier
This is the resource_identifier value of the requested resource, complete with current version number.

session_nb
This is the session number allocated by the host for the new session.

7.2.6.4 Create Session Response

This object is issued by the module providing a resource to the host in order to tell the module whether the session could be opened.

Table 9: Create Session Response coding

Syntax	No. of bits	Mnemonic
create_session_response () {		
create_session_response_tag	**8**	**uimsbf**
length_field() = 7		
session_status	**8**	**uimsbf**
resource_identifier()		
session_nb	**16**	**uimsbf**
}		

session_status values
The same status values are used as for open_session_response - see Table 7.7.

resource_identifier
The module inserts in the create_session_response the class and type of the resource requested but with the version number that the module currently supports.

session_nb
This has the same value as the session_nb field in the create_session object to which this is a reply.

7.2.6.5. Close Session Request

This object is issued by the module or by the host to close a session.

Table 10: Close Session Request coding

Syntax	No. of bits	Mnemonic
close_session_request () {		
close_session_request_tag	**8**	**uimsbf**
length_field() = 2		
session_nb	**16**	**uimsbf**
}		

7.2.6.6 Close Session Response

This object is issued by the module or by the host to acknowledge the closing of the session.

Table 11: Close Session Response coding

Syntax	No. of bits	Mnemonic
close_session_response () {		
close_session_response_tag	**8**	**uimsbf**
length_field() = 3		
session_status	**8**	**uimsbf**
session_nb	**16**	**uimsbf**
}		

session_status values

Table 12: Close Session Status values

session status	session_status value (hex)
session is closed as required	00
session_nb in the request is not allocated	F0
other	reserved

7.2.6.7. Session Number

This object always precedes a body of the SPDU containing APDU(s).

Table 13: Session Number coding

Syntax	No. of bits	Mnemonic
session_number () {		
session_number_tag	**8**	**uimsbf**
length_field() = 2		
session_nb	**16**	**uimsbf**
}		

7.2.7 Coding of the session tags

Table 14 below gives the names of the objects used by the session layer on the command interface. The coding of the spdu_tag follows the ASN.1 rules. Each spdu_tag is coded in one byte.

Table 14: Session Tag Values

spdu_tag	tag value (hex)	Primitive or Constructed	Direction host <-->module
$T_{open\_session\_request}$	'91'	P	<---
$T_{open\_session\_response}$	'92'	P	--->
$T_{create\_session}$	'93'	P	--->
$T_{create\_session\_response}$	'94'	P	<---
$T_{close\_session\_request}$	'95'	P	<--->
$T_{close\_session\_response}$	'96'	P	<--->
$T_{session\_number}$	'90'	P	<--->

Other values in the ranges 80-8F, 90-9F, A0-AF, B0-BF are reserved.

8 Command Interface - application layer

8.1 Introduction

The application layer implements a set of protocols based upon the concept of a *resource*. A resource defines a unit of functionality which is available to applications running on a module. Each resource supports a set of objects and a protocol for interchanging them to use the resource. Communication with a resource is by means of a session created to that particular resource. This clause describes the minimum set of resources that all hosts conformant to this specification shall provide. Note that some resources in clause 8 have "DVB" as a prefix in their name. These make use of DVB-specific features which may not be present in all possible situations where this interface may be used. The DVB-prefixed resources must be provided in any DVB-compliant host and may optionally be provided in other hosts. Other optional resources are described in annex B to this specification and may be described in further annexes.

8.2 Resources

8.2.1 Introduction

A resource may be provided by the host directly or it may reside in another module. A resource is identified by a resource identifier which comprises three components: resource class, resource type and resource version. Resource class defines a set of objects and a protocol for using them. Resource type defines distinct resource units within a class. All resource types within a class use the same objects and protocol but offer different services or are different instances of the same service. Resource version allows the host to identify the latest version (highest version number) of a resource where more than one of the same class and type are present. This allows updated or enhanced resources to be supplied on a module to supersede existing resources in the host or on another module. Resources with higher version numbers shall be backwards compatible with previous versions, so that applications requesting a previous version will have a resource with expected behaviour.

Resources are used by an application creating a session to a resource (see 7.2). By an initialisation process carried out by the Resource Manager the host will have identified all available resources, whether self-provided or provided on another module, and can complete the session to the appropriate place. Once the session has been created the application can then use the resource by an exchange of objects according to the defined protocol.

An example of the use of the resource concept is the Low-Speed Communications resource class (see 8.7.1). This provides a common mechanism for using a serial communications link - typically a modem or a return channel on a cable system. Resource types are defined for different speeds of modem as identified by ITU-T recommendation numbers - e.g. V21, V.22, V.32bis. Most modern modems offer a range of speeds so that one modem may exhibit several resource types and the speed selection is made by creating a session to the particular resource type which offers it.

8.2.2 Resource identifier

A resource identifier consists of 4 octets. The two most significant bits of the first octet indicate whether the resource is public or private and hence the structure of the remainder of the field. Values of 0,1 and 2 indicate a public resource. A value of 3 indicates a private resource.

Public resource classes have values allocated in the range 1 to 49150, treating the resource_id_type field as the most significant part of resource_class. Value 0 is reserved. The maximum (all-ones) value of all fields is reserved. Private resources are identified by a private resource definer, that is the organisation that defines a private resource. Each private resource definer can define the structure and content of the private_resource_identity field in any way he chooses except that the maximum (all-ones) value is reserved.

Table 15: resource_identifier coding

Syntax	No. of bits	Mnemonic
resource_identifier() {		
resource_id_type	**2**	**uimsbf**
if (resource_id_type != 3) {		
resource_class	**14**	**uimsbf**
resource_type	**10**	**uimsbf**
resource_version	**6**	**uimsbf**
}		
else {		
private_resource_definer	**10**	**uimsbf**
private_resource_identity	**20**	**uimsbf**
}		
}		

8.2.3 Applications and resource providers

These are the two types of application layer entity that can reside on a module. Applications make use of resources to perform tasks for the user of the host. Resource providers provide resources additional to those available directly in the host, or a newer version of a resource replacing one previously provided by the host, or on another module. To avoid deadlock problems and complexities of initialisation resource providers shall not be dependent on the presence of any other resources, except the Resource Manager, to provide the resources they offer.

8.3 Application Protocol Data Units

8.3.1 Introduction

All protocols in the Application Layer use a common Application Protocol Data Unit (APDU) structure to send application data between module and host or between modules.

The APDU structure is illustrated by figure 12 and table 16 below :

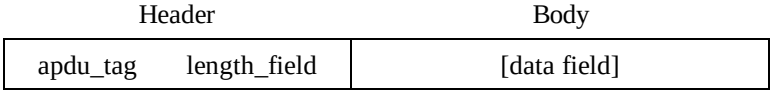

Figure 12 - APDU structure

Table 16: APDU coding

Syntax	No. of bits	Mnemonic
APDU() {		
apdu_tag	**24**	**uimsbf**
length_field()		
for (i=0;i<length_value;i++) {		
data_byte	**8**	**uimsbf**
}		
}		

The APDU is made of two parts :

- a mandatory header made of a Tag value apdu_tag, indicating which parameter is sent within the data field and a length value, coding the length of the following data field. apdu_tag and length_field are coded according to the basic encoding rules of ASN.1 [3].
- a conditional body of variable length equal to the length coded by length_field.

The APDU bodies are described in the following subclauses. This list may be extended in future versions.

8.3.2 Chaining of APDU data fields

The data field which is in an APDU may be split into several blocks of smaller sizes if required by the transmission and reception buffer sizes of the host or of the module. This mechanism is available for a few APDUs only. The chaining is performed by using two different apdu_tag values (M_apdu_tag and L_apdu_tag). All blocks of data field, except the last one is sent within a APDU with a M_apdu_tag value. This tag value indicates that more data will be sent in another APDU. The last block of data field is sent within a APDU with a L_apdu_tag value. When the last block is received, the receiving entity concatenates all the received data fields.

This mechanism is valid for all APDUs with two defined tag values (M_apdu_tag and L_apdu_tag). It is illustrated in figure 13 below.

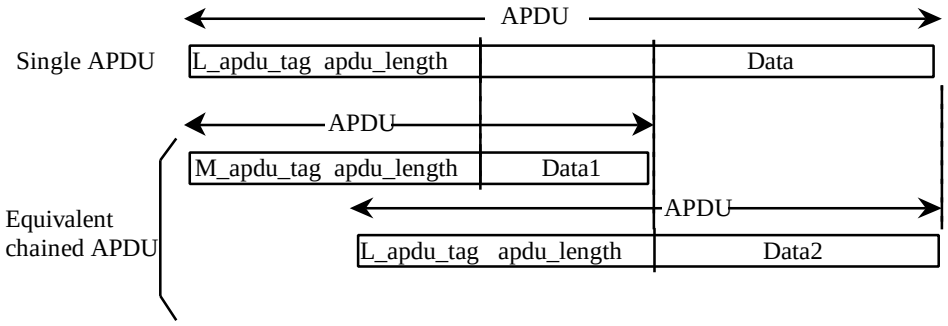

Figure 13: Illustration of the chaining mechanism

8.3.3 Transportation of APDU within SPDU

An integer number of APDUs belonging to the same session can be transported in the body of one SPDU.

8.4 System Management Resources

8.4.1 Resource Manager

The Resource Manager is a resource provided by the host. There is only one type in the class and it can support any number of sessions. It controls the acquisition and provision of resources to all applications. A symmetrical communication protocol is defined between the module and the host to determine the resources each can provide. The protocol is used first by the host to interrogate each transport connection in turn to determine what resources, if any, are presented for use on that transport connection. Then it is used by applications to find out

the total resources available. It is then used periodically when resources change to update the common view of available resources. The Resource Manager is provided by the host and cannot be superseded by a resource on a module. Any attempt to provide a Resource Manager resource by a module shall be ignored by the host.

8.4.1.1 Resource Manager Protocol

When a module is plugged in or the host is powered up one or perhaps two transport connections are created to the module, serving an application and/or a resource provider. The first thing an application or resource provider does is to request a session to the Resource Manager resource, which is invariably created since the Resource Manager has no session limit. The Resource Manager then sends a Profile Enquiry to the application or resource provider which responds with a Profile Reply listing the resources it provides (if any). The application or resource provider must now wait for a Profile Change object. Whilst waiting for Profile Change it can neither create sessions to other resources nor can it accept sessions from other applications, returning a reply of 'resource non-existent' or 'resource exists but unavailable' as appropriate.

When it has asked for profiles on all transport connections and received Profile Replies the host builds a list of available resources. Where two or more resources match in both class and type the host keeps the one with the highest version number in its list. Where the version numbers match also the host keeps all resources and chooses one at random when a create session request is received for it. Once the host has built its resource list it sends a Profile Change object on all current Resource Manager sessions, and those applications that wish to can then ask the host for its list of resources using the Profile Enquiry object.

When it receives the Profile Change notification for the first time the application or resource provider can interrogate the host with a Profile Enquiry and receive a Profile Reply with the host's list of available resources. After this first operation of the Profile Change protocol the application or resource provider is now free to create or accept other sessions. Its session to the Resource Manager persists to allow further Profile Change notification by the host from time to time.

If a resource provider wishes to notify a change in the profile of resources it provides, it issues a Profile Change to the host. The host replies with a Profile Enquiry to which the resource provider replies in turn with its updated resource list. The host processes this and, if this results in any change to the host's own resource list, the host will issue a Profile Change on all active Resource Manager sessions. The applications can then enquire and receive an updated resource list if they wish.

8.4.1.2 Profile Enquiry

The Profile Enquiry object requests the recipient to reply with a list of the resources it provides in a Profile Reply object.

Table 17: Profile Enquiry object coding

Syntax	**No. of bits**	**Mnemonic**
profile_enq () {		
profile_enq_tag	**24**	**uimsbf**
length_field()=0		
}		

8.4.1.3. Profile Reply

This is sent in response to a profile enquiry and lists the resources that the sender is able to provide. Resource identifiers for the minimum set of resources which shall be provided are listed in 8.8. Further, optional resources which have been defined may be listed in annexes to this specification.

Table 18: Profile Reply object coding

Syntax	No. of bits	Mnemonic
profile_reply () {		
profile_reply_tag	**24**	**uimsbf**
length_field() = N*4		
for (i=0; i<N; i++) {		
resource_identifier()		
}		
}		

8.4.1.4. Profile Changed

The Profile Changed object notifies the recipient that a resource has changed. A module would typically use it to notify the host if the availability status of any of its resources had changed (but not just if a resource was in use). The host would modify its own resource list if necessary, and if there was any change it would in turn send a Profile Changed object on all transport connections.

Table 19: Profile Changed object coding

Syntax	No. of bits	Mnemonic
profile_changed () {		
profile_changed_tag	**24**	**uimsbf**
length_field()=0		
}		

8.4.2 Application Information

The Application Information resource enables applications to give the host a standard set of information about themselves. Like the Resource Manager it is provided only by the host and has no session limit. All applications create a session to this resource as soon as they have completed their Profile Enquiry phase of initialisation. The host then sends an Application Info Enquiry object to the application, which responds by returning an Application Info object with the appropriate information. The session is maintained so that the host can at any time signal the application to create a MMI session at its top-level menu entry point. This is done with the Enter Menu object. When the application receives the Enter Menu object it shall create a MMI session (see 8.6) and display its top-level menu. The host shall guarantee that such a session is available to be created when it sends the Enter Menu object.

8.4.2.1 Application Info Enquiry

Table 20: Application Info Enquiry object coding

Syntax	No. of bits	Mnemonic
application_info_enq () {		
application_info_enq_tag	**24**	**uimsbf**
length_field()=0		
}		

8.4.2.2. Application Info

Table 21: Application Info object coding

Syntax	No. of bits	Mnemonic
application_info () {		
application_info_tag	**24**	**uimsbf**
length_field()		
application_type	**8**	**uimsbf**
application_manufacturer	**16**	**uimsbf**
manufacturer_code	**16**	**uimsbf**
menu_string_length	**8**	**uimsbf**
for (i=0; i<menu_string_length; i++) {		
text_char	**8**	**uimsbf**
}		
}		

application_type

application_type	application_type value
Conditional_Access	01
Electronic_Programme_Guide	02
reserved	other values

application_manufacturer
Values for this field are derived from the CA System ID values defined in [5].

manufacturer_code
The content of this field is defined by each manufacturer as he wishes.

menu_string_length
All applications have a user menu tree, of which this is the top-level entry point, and it is made available as a subtree somewhere in the host's own menu tree. It is followed by a sequence of characters which is the title of the menu entry. The host is free to decide the structure of its own menu tree but it may use the application_type field to group menu entries of similar applications. The 'menu' may in fact be a simple display with no user interaction, or it may be a complex set of menu screens to allow sophisticated user interaction.

text_char
Text information is coded using the character sets and methods described in [4].

8.4.2.3 Enter Menu

Table 22: Enter Menu object coding

Syntax	No. of bits	Mnemonic
enter_menu () {		
enter_menu_tag	**24**	**uimsbf**
length_field()=0		
}		

8.4.3 Conditional Access Support

This resource provides a set of objects specifically to support Conditional Access applications. Like the Resource Manager it is provided only by the host and has no session limit. All CA applications create a session to this resource as soon as they have completed their Application Information phase of initialisation. The host sends a CA Info Enquiry object to the application, which responds by returning an CA Info object with the appropriate information. The session is then kept open for periodic operation of the protocol associated with the CA PMT and CA PMT Reply objects.

8.4.3.1 CA Info Enquiry

Table 23: CA Info Enquiry object coding

Syntax	No. of bits	Mnemonic
ca_info_enq () {		
ca_info_enq_tag	**24**	**uimsbf**
length_field()=0		
}		

8.4.3.2 CA Info

Table 24: CA Info object coding

Syntax	No. of bits	Mnemonic
ca_info () {		
ca_info_tag	**24**	**uimsbf**
length_field()		
for (i=0; i<N; i++) {		
CA_system_id	**16**	**uimsbf**
}		
}		

CA_system_id
This lists the CA system IDs supported by this application. Values for CA System IDs are defined in [5].

8.4.3.3 Selection of services to be descrambled

CA PMT is sent by the host to one or several connected CA applications in order to indicate which elementary streams are selected by the user and how to find the corresponding ECMs. Each CA PMT object contains references to selected elementary streams of one selected programme. If several programmes are selected by the user, then several CA PMT objects are sent. The host may decide to send the CA PMT to all connected CA applications or preferably only to the applications supporting the same CA_system_id value as the value given in the CA_descriptor of the selected elementary streams (ES).

Each application then answers, when requested by the host, with the CA PMT Reply which allows the host to select the module that will perform the descrambling.

8.4.3.4 CA_PMT

The CA PMT object is a table extracted from the Programme Map Table (PMT) in the PSI information (see [1] subclauses 2.4.4.8 and 2.4.4.9) by the host and sent to the application. This table contains all access control information allowing the application to filter the ECMs itself and to make itself the correct assignment of an ECM stream with a scrambled component.

Table 25: CA PMT object coding

Syntax	No. of bits	Mnemonic
ca_pmt () {		
ca_pmt_tag	**24**	**uimsbf**
length_field()		
ca_pmt_list_management	**8**	**uimsbf**
program_number	**16**	**uimsbf**
reserved	**2**	**bslbf**
version_number	**5**	**uimsbf**
current_next_indicator	**1**	**bslbf**
reserved	**4**	**bslbf**
program_info_length	**12**	**uimsbf**
if (program_info_length != 0) {		
ca_pmt_cmd_id /* at program level */	**8**	**uimsbf**
for (i=0; i<n; i++) {		
CA_descriptor() /* CA descriptor at programme level */		
}		
}		
for (i=0; i<n; i++) {		
stream_type	**8**	**uimsbf**
reserved	**3**	**bslbf**
elementary_PID /* elementary stream PID */	**13**	**uimsbf**
reserved	**4**	**bslbf**
ES_info_length	**12**	**uimsbf**
if (ES_info_length != 0) {		
ca_pmt_cmd_id **/***at ES level ***/**	**8**	**uimsbf**
for (i=0; i<n; i++) {		
CA_descriptor() /* CA descriptor at elementary stream level */		
}		
}		
}		
}		

The CA PMT contains all the CA_descriptors of the selected programme. If several programmes are selected, the host sends several CA PMT objects to the application. The CA_PMT only contains CA_descriptors. All other descriptors must be removed from the PMT by the host.

Except for ca_pmt_list_management and ca_pmt_cmd_id which are described below, all fields of the CA PMT are described in [1] subclauses 2.4.4.8 and 2.4.4.9.

The CA_descriptor() used in this description is the CA_descriptor() defined by [1] subclause 2.6.16.

The CA_descriptor after the current_next_indicator is at the programme level and is valid for all elementary components of the programme. The CA_descriptor(s) at elementary_stream level is (are) valid for the elementary_stream only. If, for one elementary_stream, CA_descriptor(s) exist at programme level and at elementary_stream level, only the CA_descriptor(s) at elementary_stream level are taken into account.

The receiver sends a new CA PMT or a new list of CA PMT to the Application when :

- the user selects another programme
- a 'tune' command selects another service (see 8.5.1.1)
- the version_number changes
- the current_next_indicator changes

ca_pmt_list_management

This parameter is used to indicate whether the user has selected a single programme (made of one or several elementary_streams) or several programmes. The following values can be used:

ca_pmt_list_management	value
more	00
first	01
last	02
only	03
add	04
update	05
reserved	other values

When set to 'first' it means that the CA PMT is the first one of a new list of more than one CA PMT object. All previously selected programmes are being replaced by the programmes of the new list. 'more' means that the CA PMT is neither the first one, nor the last one of the list. 'last' means that the CA PMT object is the last of the list. 'only' means that the list is made of a single CA PMT. 'add' means that this CA PMT has to be added to an existing list, that is, a new programme has been selected by the user, but all previously selected programmes remain selected. If 'add' is received for an already existing programme then its action is identical to 'update'. 'update' means that the CA PMT of a programme already in the list is sent again because the version_number or the current_next_indicator has changed. It is the responsibility of the application to check whether the change of the version_number results in a change of the CA operations or not. Since the list management commands only act at the programme level, any changes at the elementary stream level in an existing programme must be signalled by an 'update' command with the complete elementary stream list resent.

ca_pmt_cmd_id

This parameter indicates what response is required from the application to a CA PMT object. It can take the following values:

ca_pmt_cmd_id	ca_pmt_cmd_id value
ok_descrambling	01
ok_mmi	02
query	03
not_selected	04
RFU	other values

When set to 'ok_descrambling' it means that the host does not expect answer to the CA PMT and the application can start descrambling the programme or start an MMI dialogue immediately. When set to 'ok_mmi' it means that the application can start a MMI dialogue but shall not start descrambling before reception of a new CA PMT object with ca_pmt_cmd_id set to 'ok_descrambling'. In this case the host shall guarantee that a MMI session can be opened by the CA application. When set to 'query' it means that the host expects to receive a CA PMT Reply. In this case, the application is not allowed to start descrambling or MMI dialogue before reception of a new CA PMT object with the ca_pmt_cmd_id set to 'ok_descrambling' or to 'ok_mmi'. When set to 'not_selected' it indicates to the CA application that the host no longer requires that CA application to attempt to descramble the service. The CA application shall close any MMI dialogue it has opened.

8.4.3.5 CA PMT Reply

This object is always sent by the application to the host after reception of a CA PMT object with the ca_pmt_cmd_id set to 'query'. It may also be sent after reception of a CA PMT object with the ca_pmt_cmd_id set to 'ok_mmi' in order to indicate to the host the result of the MMI dialogue ('descrambling_possible' if the user has purchased, 'descrambling not possible (because no entitlement)' if the user has not purchased).

Table 26: CA PMT Reply object coding

Syntax	**No. of bits**	**Mnemonic**
ca_pmt_reply () {		
ca_pmt_reply_tag	**24**	**uimsbf**
length_field()		
program_number	**16**	**uimsbf**
reserved	**2**	**bslbf**
version_number	**5**	**uimsbf**
current_next_indicator	**1**	**bslbf**
CA_enable_flag	**1**	**bslbf**
if (CA_enable_flag ==1)		
CA_enable **/*** at programme level ***/**	**7**	**uimsbf**
else if (CA_enable_flag ==0)		
reserved	**7**	**bslbf**
for (i=0; i<n; i++) {		
reserved	**3**	**bslbf**
elementary_PID /* elementary stream PID */	**13**	**uimsbf**
CA_enable_flag	**1**	**bslbf**
if (CA_enable_flag == 1)		
CA_enable **/*** at elementary stream level ***/**	**7**	**uimsbf**
else if (CA_enable_flag ==0)		
reserved	**7**	**bslbf**
}		
}		

The syntax contains one possible ca_enable at programme level and, for each elementary stream, one possible ca_enable at elementary stream level.

- When both are present, only ca_enable at ES level applies for that elementary stream
- When none is present, the host does not interpret the ca_pmt_reply object.

The CA_enable field indicates whether the application is able to perform the descrambling operation. CA_enable values are coded as follows:

CA_enable	**value**
Descrambling possible	01
Descrambling possible under conditions (purchase dialogue)	02
Descrambling possible under conditions (technical dialogue)	03
Descrambling not possible (because no entitlement)	71
Descrambling not possible (for technical reasons)	73
RFU	other values 00-7F

The value "descrambling possible" means that the application can descramble with no extra condition (e.g. : the user has a subscription) or that the user has authorised the purchase of the elementary stream.

The value "descrambling possible under conditions (purchase dialogue)" means that the application has to enter into a purchase dialogue with the user before being able to descramble (pay per view programme).

The value "descrambling possible under conditions (technical dialogue)" means that the application has to enter into a technical dialogue with the user before being able to descramble (e.g. ask the user to select fewer elementary streams because the descrambling capabilities are limited).

The value "descrambling not possible (because no entitlement)" means that the user has no entitlement for the programme or the user does not want to buy the programme.

The value "descrambling not possible (for technical reasons)" means that the application cannot descramble the elementary stream for technical reasons (e.g. : all descrambling capabilities are already in use).

The protocol allows services to be selected for descrambling at either the programme level or the elementary stream level. Where the host cannot support descrambling of different elementary streams by different modules, then it shall take as the CA_enable value for the programme the *lowest* of the CA_enable values returned for each elementary stream of the programme. This ensures that any stream will be descrambled if it can be.

8.5 Host Control and Information Resources

8.5.1. DVB Host Control

The DVB Host Control resource allows an application limited control over the operation of the host. In this version there are two major functions. The first is to re-tune the host to another service. This may change frequencies - it may even steer to a different satellite. Context of the previous state may not be retained. The second function is to temporarily replace one service by another. This may be of short duration, such as advertisement replacement, or of longer duration, such as a whole programme. The first function is supported by a Tune object. The second is supported by Replace, and Cancel Replace objects. There is also an Ask Release object for the host to ask an application to close the session. Only the host provides the DVB Host Control resource, and it can only support one session at a time.

8.5.1.1 Tune

This allows the application to have the host tune to a different service. The parameters of this object, Network ID, Original Network ID, Transport Stream ID and Service ID are defined in [4]. When the host has tuned to the new service it must enter into the standard CA support dialogue using the CA_PMT object (see 8.4.4) to enable the new service to be descrambled. Sending this object may cause the host to lose its current state. If so, the host will not re-tune to the previous service when the DVB Host Control session is closed.

Table 27: Tune object coding

Syntax	No. of bits	Mnemonic
tune () {		
tune_tag	**24**	**uimsbf**
length_field()		
network_id	**16**	**uimsbf**
original_network_id	**16**	**uimsbf**
transport_stream_id	**16**	**uimsbf**
service_id	**16**	**uimsbf**
}		

8.5.1.2 Replace

Replace and Clear Replace allow one service component to be temporarily replaced by another from the same multiplex. The application allocates a replacement_ref value which is used to match a Clear Replace object with one or more previous Replace objects. The application then sends one or more Replace objects to request that a component in the replaced_PID - video, audio, teletext or subtitles - be replaced by the component being transmitted in the replacement_PID. The replacement occurs immediately. Several Replace objects can use the same value of replacement_ref, in which case they will all be cleared when the appropriate Clear Replace object is sent. The host shall retain the context for the previous services so that it can reinstate them when Clear Replace is sent. The previous context is also restored, if possible, when the session to the DVB Host Control resource is closed.

Table 28: Replace object coding

Syntax	No. of bits	Mnemonic
replace () {		
replace_tag	**24**	**uimsbf**
length_field()		
replacement_ref	**8**	**uimsbf**
reserved	**3**	**bslbf**
replaced_PID	**13**	**uimsbf**
reserved	**3**	**bslbf**
replacement_PID	**13**	**uimsbf**
}		

8.5.1.3 Clear Replace

Table 29: Clear Replace object coding

Syntax	No. of bits	Mnemonic
clear_replace () {		
clear_replace_tag	**24**	**uimsbf**
length_field()		
replacement_ref	**8**	**uimsbf**
}		

8.5.1.4 Ask Release

Whilst an application has a session in progress to the DVB Host Control resource, it has control over aspects of the host's behaviour. It may be necessary for the host to regain control and in order to do this it sends the Ask Release object to the application. The application is given a short time-out period to send Clear Replace objects, if necessary, and close the session. If the session is not closed at the end of the time-out period the host will close it anyway and restore the previous host state, if possible.

Table 30: Ask Release object coding

Syntax	No. of bits	Mnemonic
ask_release () {		
ask_release_tag	**24**	**uimsbf**
length_field() = 0		
}		

8.5.2 Date and Time

The date and time resource is supplied by the host and it can support unlimited sessions. An application creates a session to the resource and then enquires the current time with a Date-Time Enquiry object. If response_interval is zero, then the response is a single date_time object immediately. If response_interval is non-zero then the response is a date_time object immediately, followed by further date_time objects every response_interval seconds. In a DVB-compliant host the supplied time is derived from the Time and Date Table (TDT) in the SI information (see [4]).

8.5.2.1 Date-Time Enquiry

Table 31: Date-Time Enquiry object coding

Syntax	No. of bits	Mnemonic
date_time_enq () {		
date_time_enq_tag	**24**	**uimsbf**
length_field()		
response_interval	**8**	**uimsbf**
}		

8.5.2.2 Date-Time

Table 32: Date-Time Enquiry object coding

Syntax	No. of bits	Mnemonic
date_time () {		
date_time_tag	**24**	**uimsbf**
length_field()= 5 or 7		
UTC_time	**40**	**bslbf**
local_offset /* optional */	**16**	**tcimsbf**
}		

UTC_time
This carries the UTC date and time to the nearest second coded as Modified Julian Day and hours, minutes and seconds as described in [4].

local_offset
This is an optional field. If present it codes the current offset between UTC and local time as a signed number of minutes. Local Time = UTC_time + local_offset. The host shall provide the local_offset field when it has a reliable knowledge of its value. Otherwise it shall omit it.

8.6 Man-Machine Interface Resource

8.6.1 Introduction

One resource classes is provided. It supports display and keypad interactions with the user. Two levels of interaction are defined. The Low-Level MMI mode gives the application detailed control over aspects of the interaction, including receiving remote control key codes direct from the user, and being able to control in detail the layout and other attributes of the screen display. The High-Level MMI mode gives the application a set of objects with higher-level semantics, including menus and lists. In this case the host determines the look and feel of the display. It is not possible to mix MMI modes in the same session. Whether the host can support more than one MMI session simultaneously is a host design choice.

Text delivered by display objects in high level mode is coded according to [4]. Text delivered by display objects in low level are coded according to [9].

8.6.2 Objects used in both modes

8.6.2.1 Close MMI

This object may be sent by host or module to allow the user or an application to leave any dialogue in high- or low-level MMI mode.

Table 33: Close MMI object coding

Syntax	No. of bits	Mnemonic
close_mmi () {		
close_mmi_tag	**24**	**uimsbf**
length_field()		
close_mmi_cmd_id	**8**	**uimsbf**
if (close_mmi_cmd_id == delay) {		
delay	**8**	**uimsbf**
}		
}		

close_mmi_cmd_id

close_mmi_cmd_id	value
immediate	00
delay	01
reserved	other values

Indicates whether the display should be returned immediately to the previous service or whether it should be delayed to allow another MMI dialogue to replace this one.

delay
The delay, in seconds, before the display should be returned to the current service, if another MMI session has not started in the meantime.

When sent by the application the host shall immediately close the current MMI session to the application. If close_mmi_cmd_id is 'immediate' then the host shall also return to its previous display state immediately. If close_mmi_cmd_id is 'delay' then the host shall maintain the last screen state of the MMI session until either the specified delay expires or until another MMI session is started. The intention is to allow seamless MMI dialogue changes between one application and another, without brief bursts of video from the current service intervening. If the application closes the session to the MMI resource rather than using this object, the host shall interpret it as a close_mmi with the 'immediate' command.

When sent by the host the application shall close the current MMI session. It is done this way in preference to just having the host close the session to the MMI resource in order to allow the application to close down gracefully. In this case the delay function is unnecessary and the host shall always use the 'immediate' command.

8.6.2.2 Display Control

The display control object allows an application to ask the host about its graphics/display capabilities under 3 modes of operation:
- bit map graphics overlaid over video
- bit map graphics replacing video
- character based high level MMI mode

The profiling communications allow the application to understand:
- the addressable co-ordinate system of the display (principally to distinguish 625 and 525 line displays)
- the aspect ratio of the pixels
- the number of pixels/pixel depths that the application can use
- character sets available

The display control object performs two functions. One is to request information about the information display characteristics of the display. The second is to set the mode of the display for information display.

Table 34: Display Control object coding

Syntax	No. of bits	Mnemonic
display_control() {		
display_control_tag	**24**	**uimsbf**
length_field()		
display_control_cmd	**8**	**uimsbf**
if (display_control_cmd == set_MMI_mode) {		
MMI_mode	**8**	**uimsbf**
}		
}		

The display control commands are defined below:

display_control_cmd	value	description of command
set_mmi_mode	01	Request that host enters the MMI mode indicated by the MMI_mode byte
get_display_character_table_list	02	Request that the host return a list of the character code tables that it can support during display operations.
get_input_character_table_list	03	Request that the host return a list of the character code tables that it can support during input operations.
get_overlay_graphics_characteristics	04	Request the profile of the display when used to display graphics overlaid over video.
get_full-screen_graphics_characteristics	05	Request the profile of the display when used to display graphics in replacement of video.
reserved	other values	

mmi_mode	value	description
high level	01	Request that a high level MMI session is opened. If implemented on the main video display this may partially or completely obscure any video that is currently being displayed.
low level overlay graphics	02	Request that a graphical low level MMI session is opened overlaying the main video display (if one is active).
low level full screen graphics	03	Request that a graphical low level MMI session is opened replacing (or independent of) the main video display.
reserved	other values	

8.6.2.3 Display Reply

The display reply object is the response by the host's display system to the display control objects sent by the application.

Table 35: Display Reply object coding

Syntax	No. of bits	Mnemonic
display_reply() {		
display_reply_tag	**24**	**uimsbf**
length_field()		
display_reply_id	**8**	**uimsbf**
if (display_reply_id == list_graphic_overlay_characteristics ||		
display_reply_id ==list_full_screen_graphic_characteristics) {		
display_horizontal_size	**16**	**uimsbf**
display_vertical_size	**16**	**uimsbf**
aspect_ratio_information	**4**	**uimsbf**
graphics_relation_to_video	**3**	**bslbf**
multiple_depths	**1**	**bslbf**
display_bytes	**12**	**uimsbf**
composition_buffer_bytes	**8**	**uimsbf**
object_cache_bytes	**8**	**uimsbf**
number_pixel_depths	**4**	**uimsbf**
for (i=0;i<n;i++) {		
display_depth	**3**	**uimsbf**
pixels_per_byte	**3**	**uimsbf**
reserved	**2**	**bslbf**
region_overhead	**8**	**uimsbf**
}		
if (display_reply_id == list_display_character_tables ||		
display_reply_id == list_input_character_tables) {		
for (i=0;i<n;i++) {		
character_table_byte	**8**	**uimsbf**
}		
}		
if (display_reply_id == mmi_mode_ack) {		
/* acknowledge of the selected mmi mode */		
mmi_mode	**8**	**uimsbf**
}		
}		

display_reply_id
This is coded as follows:

display_reply_id	id value
mmi_mode_ack	01
list_display_character_tables	02
list_input_character_tables	03
list_graphic_overlay_characteristics	04
list_full_screen_graphic_characteristics	05
unknown display_control_cmd	F0
unknown_mmi_mode	F1
unknown_character_table	F2
reserved	other values

display_horizontal_size
display_vertical_size
These 16 bit integers describe the maximum addressable co-ordinate range of the bit mapped graphic display. All positions from (0, 0) to (display_horizontal_size-1, display_vertical_size- 1) can be addressed. All decoders should support at least 720 pixel columns. The number of scan lines provided should be at least that of the vertical resolution of the video format being decoded at that time. So, when working with 625/50 video at least 576 lines should be provided and at least 480 lines should be provided when working with 525/60 video.

aspect_ratio_information
This 4 bit field is coded in the same way as the aspect_ratio_information field of ISO/IEC 13818-2. It allows the pixel aspect ratio to be determined from the display_horizontal_size, display_vertical_size.

graphics_relation_to_video
This 3 bit field indicates the extent to which the video plane corresponds to the graphics plane. This will indicate whether graphics can be predictability positioned relative to the video. The broadcaster should be aware that any video broadcast at reduced resolution may be scaled by the post processing of the video decoder in an implementation dependent way. In the limiting case the graphics could be presented on a different display device to the video.

graphics_relation_to_video	value
No relationship between graphics and video.	000
reserved	001 - 110
The graphics co-ordinate space exactly matches the video co-ordinate space.	111

multiple_depths
When set to '1' this 1 bit field indicates whether the display can support a mixture of different pixel depths (i.e. each display region could be specified with a different pixel depth). When set to '0' this field indicates that the display system has (unspecified) restrictions on mixing different pixels depths. In this case only a single pixel depth should be used for all regions.

display_bytes
This 12 bit integer when multiplied by 256 gives the number of bytes available for the display memory.

composition_buffer_bytes
This 8 bit integer when multiplied by 256 gives the number of bytes available for the composition buffer.

object_cache_bytes
This 8 bit integer when multiplied by 4096 gives the number of bytes available for the object cache buffer. The memory requirement for each object is that for the DVB_Subtitling_segment that delivers the object.

number_pixel_depths
This 4 bit integer indicates the number of different pixel depths that the display can provide.

display_depth
This 3-bit field indicates the display pixel depth. It is coded in the same manner as region_level_of_-compatibility in [9].

pixels_per_byte
This 3 bit integer gives the number of pixels that can be packed into each byte at this display depth. The value 0 is a special case implying an 8 bit deep display.

region_overhead
This 8 bit integer when multiplied by 16 gives the reduction in the number of pixels that can be displayed when an additional region is introduced at this pixel depth.

The decoders should be able to implement the display requested by the application when the following condition is true:

display_bytes >= bytes_used

Where bytes used is calculated by the algorithm shown below:

```
bytes_used = 0;
for (depth=0; depth < number_pixel_depth; depth++) {
   for (region=0; region < number of regions with pixel depth == depth;
        region++) {
      bytes_used += (region_width(region, depth) *
         region_height(region, depth)) / pixels_per_byte(depth);
      bytes_used += region_overhead(depth);
   }
}
```

character_table_byte
In [4] character table selection for text fields (if different from the default) is indicated by the initial byte (or bytes) of the text field. The character table bytes are a non-ordered list of the character table selection bytes defined by [4].

All hosts must support input and output using the default (table 0) Latin Alphabet as defined by [4].

8.6.3 Low-Level MMI Keypad objects

The objects below are used in low level MMI mode only.

8.6.3.1 Keypad Control

The application makes the assumption that the host has a keypad and that the user uses this keypad in the man machine dialogue. This keypad may be *virtual* : the user may enter commands or choices in another way (e.g. : voice) and the host translates these user actions into virtual keypresses for the application.

Table 36: Keypad Control object coding

Syntax	No. of bits	Mnemonic
keypad_control() {		
keypad_control_tag	**24**	**uimsbf**
length_field()		
keypad_control_cmd	**8**	**uimsbf**
if (keypad_control_cmd == intercept_selected_keypress) {		
for (i=0;i<keypad_control_length - 1;i++){		
/* list of accepted keypresses */		
key_code	**8**	**uimsbf**
}		
}		
if (keypad_control_cmd == ignore_selected_keypress) {		
for (i=0;i<keypad_control_length - 1;i++){		
/* list of ignored keypresses */		
key_code	**8**	**uimsbf**
}		
}		
if (keypad_control_cmd == reject_keypress) {		
key_code /* rejected keypress */	**8**	**uimsbf**
}		
}		

The Keypad Control object and its associated keypress object described below allow the application to intercept virtual keypresses. The Keypad Control object allows the application to direct keypresses to the application. These are sent via the Keypress object when they occur. The application can also stop accepting keypresses, and also reject keypresses it does not understand or is not interested in. The set of keypresses that can be intercepted is defined in 8.6.3.3.

keypad_control_cmd	cmd value
intercept_all_keypresses	01
ignore_all_keypresses	02
intercept_selected_keypress	03
ignore_selected_keypress	04
reject_keypress	05
reserved	other values

8.6.3.2 Keypress

Table 37: Keypress object coding

Syntax	No. of bits	Mnemonic
keypress() {		
keypress_tag	**24**	**uimsbf**
length_field()=1		
key_code	**8**	**uimsbf**
}		

8.6.3.3 Table of key codes

These are key codes used on the host/Application interface. The corresponding keys are not necessarily present on the keypad, but the host has to know how to translate a corresponding user action into a key code. It is mandatory to support all the key codes.

Key code values (hexadecimal) are listed in the table below.

key code	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
meaning	0	1	2	3	4	5	6	7	8	9	menu	ESC	⇒	⇐	⇑	⇓	BS	RC

Other values (from 0x12 to 0xFF) are reserved.

8.6.4 Low-Level MMI Display objects

The low-level MMI display capability is based upon the use of the DVB Subtitling mechanism.

8.6.4.1 Delivery of DVB Subtitle data

DVB Subtitling Segments are encapsulated in DVB common interface APDUs as is described below. The sequence of DVB Subtitle segments should follow that defined by [9]. After each complete display set has been transmitted from the application to the host the application should transmit a scene end mark APDU. This APDU delimits display sets and provides the application with temporal control of the decoding process.

8.6.4.2 Segment Encapsulation

DVB Subtitling Segments are encapsulated directly in DVB common interface APDUs. The more/last format is used to allow long segments to be fragmented over multiple APDU.

Table 38: subtitle_segment object coding

Syntax	No. of bits	Mnemonic
subtitle_segment() {		
subtitle_segment_tag	**24**	**uimsbf**
length_field()		
DVB_Subtitling_segment()		
}		

8.6.4.3 Display System Messages

The display system messages alert the application to situations that require attention.

Table 39: Display Message object coding

Syntax	No. of bits	Mnemonic
display_message() {		
display_message_tag	**24**	**uimsbf**
length_field()		
display_message_id	**8**	**uimsbf**
}		

display message_id	value	When used
Display OK	00	Can optionally be sent by the host as a positive acknowldegment of any low or high level MMI object.
Diplay Error	01	An error has been detected by the display system.
Display out of memory	02	An error indicating that the host has exhuasted its composition buffer, pixel buffer or object cache memory.
DVB Subtitling syntax error	03	An error indicating that the host cannot interpret the DVB Subtitling Segments.
Undefined region referenced	04	An error indicating that the host has found a reference to a region_id that has not been introduced.
Undefined CLUT referenced	05	An error indicating that the host has found a reference to a CLUT_id that has not been introduced.
Undefined object referenced	06	An error indicating that the host has found a reference to an object_id that has not been introduced.
Object incompatible with region	07	An error indicating that the pixel depth or size of an object is not compatible with the region where it is instanced.
Unknown character referenced	08	An error indicating that a character code incompatible with the selected character table has been found.
Display characteristics changed	09	This message indicates that some characteristic of the display has changed since the display characteristics where last inspected by the application. For example, this could result following a user activity such as re configuring the TV from 16:9 to 4:3 display or in response to a change in program material (e.g. the display format of the video being decoded changes).
reserved	other values	

8.6.4.4 Temporal Control

When subtitles are broadcast the PTS in the header of the PES that encapsulates the subtitle segment controls the timing of the decode and display of successive displays of subtitling information. When used from the common interface a suite of APDU provide temporal control.

scene_end_mark

Table 40: scene_end_mark object coding

Syntax	No. of bits	Mnemonic
scene_end_mark() {		
scene_end_mark_tag	**24**	**uimsbf**
length_field()		
decoder_continue_flag	**1**	**bslbf**
scene_reveal_flag	**1**	**bslbf**
send_scene_done	**1**	**bslbf**
reserved	**1**	**bslbf**
scene_tag	**4**	**uimsbf**
}		

The application should send a scene end mark APDU after the set of subtitle segments for one display (the display set) has been sent to the decoder. The scene end mark indicates to the decoder the end of the data set and tells it what to do when it has completed decoding the set.

decoder_continue_flag
This 1 bit flag when set to '1' instructs the decoder to continue decoding subtitling data (from this MMI session). When set to '0' the decoding process should stop (after any other instructions implied by the scene end mark have been implemented).

scene_reveal_flag
This 1 bit flag when set to '1' instructs the decoder to implement the immediately preceding page composition segment. I.e. the display should change from that defined by the currently active page composition segment to that defined by the page composition segment that has been decoded most recently. When this flag is set to '0' the implementation of the page composition segment is deferred until a scene reveal with the matching scene_tag is sent.

send_scene_done
This 1 bit flag when set to '1' instructs the decoder to send a scene done APDU to the application.

scene_tag
This 4 bit integer is set by the application. Its value should increment modulo 16 for each scene end mark.

scene_done_message

Table 41 - scene_done_message coding

Syntax	No. of bits	Mnemonic
scene_done_message() {		
scene_done_message	**24**	**uimsbf**
length_field()		
decoder_continue_flag	**1**	**bslbf**
scene_reveal_flag	**1**	**bslbf**
reserved	**2**	**bslbf**
scene_tag	**4**	**uimsbf**
}		

The decoder should send a scene done message when it completes decoding a display set followed by a scene end mark with the send scene done flag set.

decoder_continue_flag
scene_reveal_flag
These 1 bit flags duplicate the state of the same flags in the scene end mark that caused this message to be sent.

scene_tag
These 4 bit integer duplicates the state of the same integer in the scene end mark that caused this message to be sent.

scene_control

Table 42: scene_control coding

Syntax	No. of bits	Mnemonic
scene_control() {		
scene_control_tag	**24**	**uimsbf**
length_field()		
decoder_continue_flag	**1**	**bslbf**
scene_reveal_flag	**1**	**bslbf**
reserved	**2**	**bslbf**
scene_tag	**4**	**uimsbf**
}		

The application can send a scene control APDU only AFTER the scene done message for the corresponding display set has been sent by the decoder.

decoder_continue_flag
This 1 bit flag when set to '1' instructs the decoder to continue decoding subtitling data. This only has any effect if the decoder continue flag was set to '0' in the scene end mark.

scene_reveal_flag
This 1 bit flag when set to '1' instructs the decoder to continue implement the new page composition segment for this scene subtitling data. This only has any effect if the scene reveal flag was set to '0' in the scene end mark.

scene_tag
This 4 bit integer indicates the scene end mark which is being operated upon. Its value should increment modulo 16 for each scene control.

8.6.4.5 Object Download

The subtitle download APDU are identical in format to the subtitle segment APDU. The APDU are constrained in that they should only carry DVB Subtitling object data segments (where the subtitle segment APDU can carry any DVB Subtitling segment).

There are several implementation options for the object chaching. However, the functionality provided to the application is equivalent to the following:
- The DVB Subtitling object data segment carried by the APDU is stored in the object cache.
- When a region references an object in the cache (with appropriate object ID and object provider flag) the DVB Subtitling segment is read from the cache and supplied to the input of the decoder.

The worst case speed of the decoder is identical to that when the segment is transferred across the common interface when it is required. However, processing load on the application is reduced and the processing load on the decoder may be reduced.

Any of the object types recognised by DVB Subtitling may be stored in the cache. However, if the object type is a character or a string of characters then the method for acquiring the glyph for each character is not addressed by this standard.

Subtitle download APDU can be sent at any time that a subtitle segment APDU could carry an object data segment.

When the MMI session is opened no data is assumed to be in the cache. When the MMI session is closed the contents of the cache are purge. There cache flush object also allows the cache to be purged.

Table 43: subtitle_download coding

Syntax	No. of bits	Mnemonic
subtitle_download() {		
subtitle_download_tag	**24**	**uimsbf**
length_field()		
DVB_Subtitling_segment()		
}		

Table 44: flush_download coding

Syntax	No. of bits	Mnemonic
flush_download() {		
flush_download_tag	**24**	**uimsbf**
length_field()		
}		

Requests that the decoder's subtitling object cache is purged.

Table 45: download_reply coding

Syntax	No. of bits	Mnemonic
download_reply() {		
download_reply_tag	**24**	**uimsbf**
length_field()		
object_id	**16**	**uimsbf**
download_reply_id	**8**	**uimsbf**
}		

The download reply object allows the host to indicate problems with an object download. Where the download has been successful there is no requirement to reply.

object_id
The ID of the downloaded object that caused the message. There the message is reporting that the segment was not an object data segment the object_id should be 0xFFFF.

download_reply_id	value
Download OK	00
Not an object data segment	01
Memory exhausted	02
reserved	other values

8.6.4.6 The Subtitling Decoder Model when addressed from the Common Interface

There is no increase in processing speed required. However, the interface is able to take advantage of faster processing speed if it is available.

When operating in overlay mode the subtitling decoder is expected to provide the same pixel buffer memory and composition buffer memory as defined in [9]. However, when addressed from the common interface the application is provided with sufficient knowledge of the characteristics of the display system to be able to ensure that no unplanned colour depth quantisation occurs. Also, any host resources additional to those required by a minimum specification DVB subtitling decoder can be taken advantage of by the application.

The full screen graphics mode is not addressed by the DVB Subtitling specification. When supporting this mode of operation this specification mandates the following host resources:

minimum pixel buffer memory	207360 bytes (in principle, sufficient to provide720 x 576 x 16 colours)
minimum composition buffer memory	12 k bytes

There is no mandated minimum requirement for object cache RAM. Where the display profile indicates that no object cache RAM is provided the object download facilities should not be used by an application.

Use of object cache

Download bit map objects are instantiated by including their object_id and an object_provider_flag = 0x01 in the region composition segment (indicating that the object provider is host resident ROM).

The worst case speed of the decoder is identical to that when each segment is transferred across the common interface when it is required. However, processing load on the application is reduced and the processing load on the decoder may be reduced by using the cache. So, this technique provides an opportunity to enhance the performance of the system.

Provision of object caching is recommended but not mandatory.

Page ID

The common interface use of DVB Subtitling requires that the decoder behaves as if the DVB Subtitling segments were inserted at the input of the "subtitle processing" block after the subtitle decoder's coded data buffer. So, this insertion point is after the page filter and hence the page ID of the DVB Subtitling segment can be ignored by the decoder and need not be set by the application.

Other subtitling services

The decoder is only assumed to have a single subtitling decoder. If the decoder is already decoding a broadcast subtitling stream when the MMI session starts the decoder can suspend the operation of the decoding the broadcast data for the duration of the MMI session. After the session is closed the decoder should, where possible, return to its previous decoding task.

8.6.5 High-Level MMI objects

The objects below are used in high level MMI mode only.

The high level MMI mode objects define the transactions required but allow the host to determine the format and type of display. The application may use control characters in text but these may be ignored by the host. The control characters can be used by the host to assist in the presentation of the menu.

The contents of any text will be interpreted according to the current character set. The presentation of the information by the host is not limited to the use of a display device.

8.6.5.1 Text

The text object is used to display a block of text on the screen. It is used as part of a higher-level object in High-Level MMI mode.

Table 46: Text object coding

Syntax	No. of bits	Mnemonic
text() {		
text_tag	**24**	**uimsbf**
length_field()		
for (i=0;i<length;i++)		
text_char	**8**	**uimsbf**
}		

A text object with text_length equal to 0 will be interpreted as a null object : nothing is displayed.

text_char
Text information is coded using the character sets and methods described in [4].

The text sent by the application may include such control characters as are defined by [4] to provide indication of how the display is to be presented. The interpretation of these control characters is at the discretion of the host.

8.6.5.2 Enq

Enq and Answ allow the application to request user input. There is a single statement, which may be a request for information such as user's PIN code. The response to the Enq is returned by the host in the Answ object. The Enq object allows the application to specify whether the user's input should be echoed back to the user by the host. For example, when entering a PIN code, the numbers entered by the user should not be displayed.

Table 47: Enq object coding

Syntax	No. of bits	Mnemonic
enq () {		
enq_tag	**24**	**uimsbf**
length_field()		
reserved	**7**	**bslbf**
blind_answer	**1**	**bslbf**
answer_text_length	**8**	**uimsbf**
for (i=0;i<enq_length-2;i++)		
text_char	**8**	**uimsbf**
}		

blind_answer : set to 1 means that the user input has not to be displayed when entered. The host has the choice of the replacement character used (star, ...).

answer_text_length : expected length of the answer. Set to hex 'FF' if unknown by the Application.

text_char
Text information is coded using the character sets and methods described in [4].

Example of ENQ :

PLEASE TYPE YOUR PIN CODE	/* text of the enquiry */
'**'**	/*blind answer */

8.6.5.3 Answ

This object is used together with the Enq object to manage user inputs.

Table 48: Answ object coding

Syntax	No. of bits	Mnemonic
answ () {		
answ_tag	**24**	**uimsbf**
length_field()		
answ_id	**8**	**uimsbf**
if (answ_id == answer) {		
for (i=0;i<length;i++)		
text_char	**8**	**uimsbf**
}		
}		

When the Application wants to receive something from the user, it sends the Enq object. The host will return the Answ object (after a possible translation of the keypress codes into text_char codes). The text_chars in the Answ object shall be coded using the same character coding scheme as that used its associated Enq object. The text field in the Answ object shall use the same signalling (defined by [4]) for character code table selection as that used in the associated Enq object.

The answ_id field is used to indicate the type of response received from the user. answ_id values are coded as follows:

answ_id	value
cancel	00
answer	01
reserved	other values

The value “answer” means that the object contains the user input (which may be of zero length). The value “cancel” means that the user wishes to abort the dialogue.

8.6.5.4 Menu

This object is used in conjunction with the Menu Answ object to manage menus in the high level MMI mode.

Table 49: Menu object coding

Syntax	No. of bits	Mnemonic
menu () {		
menu_tag	**24**	**uimsbf**
length_field()		
choice_nb	**8**	**uimsbf**
TEXT() /* title text*/		
TEXT() /* sub-title text */		
TEXT() /* bottom text */		
for (i=0;i<choice_nb;i++) /* when choice_nb != 'FF' */		
TEXT()		
}		

A menu is made of one Title, one sub-title, several choices and one bottom line. Text objects with text_length = 0 can be used (e.g. if no sub-title or no bottom text are used).

Choice_nb = 'FF' means that this field does not carry the number of choices information.

The way the host has to display the text, the menu box and select the choice is manufacturer dependent. For example, the host is free to display the choices on several pages and to manage itself the page-down and page-up functions. The manufacturer is also free to define how the user selects the choice (numerical keypad, arrows, coloured keys, voice ...).

8.6.5.5 Menu Answ

This object is used in conjunction with the menu object to return the user choice, and the with list object to indicate that the user has finished with that object.

Table 50: Menu Answ object coding

Syntax	**No. of bits**	**Mnemonic**
menu_answ () {		
menu_answ_tag	**24**	**uimsbf**
length_field() = 1		
choice_ref	**8**	**uimsbf**
}		

choice_ref : the number of the choice selected by the user. If the object was preceded by a menu object, then choice_ref = 01 corresponds to the first choice that had been presented by the application in that object (first choice text after the bottom text in the menu object) and choice_ref = 02 corresponds to the second choice text presented by the application.

choice_ref = 00 indicates that the user has cancelled the preceding menu or list object without making a choice.

Example of menu :

```
DO YOU WANT TO BUY ?                          /* title */

"JURASSIC PARK"   COST : $5.00                /* sub-title */

     1/ YES                                   /* choice 1 */

     2/ NO                                    /* choice 2 */

< REMAINING CREDIT : $47.50 >                 /* bottom line */
< THE FIRST 3 MINUTES ARE FREE OF CHARGE >
```

8.6.5.6 List

This object is used to send a list of items to be displayed (e.g. during a consultation of the entitlements). It has exactly the same syntax as the Menu object, and is used in conjunction with the Menu_answ object.

Table 51: List object coding

Syntax	No. of bits	Mnemonic
list () {		
list_tag	**24**	**uimsbf**
length_field()		
item_nb	**8**	**uimsbf**
TEXT() /* title text*/		
TEXT() /* sub-title text */		
TEXT() /* bottom text */		
for (i=0;i<item_nb;i++) /* when item_nb != 'FF' */		
TEXT()		
}		

A list is made of one Title, one sub-title, several items and one bottom line. Text objects with text_length = 0 can be used (e.g. if no sub-title or no bottom text are used).

Item_nb = 'FF' means that this field does not carry the number of items information.

The way the host has to display the title, sub-title, bottom text and items is manufacturer dependant. For example, the host is free to display the items on several pages and to manage itself the page-down and page-up functions.

8.7 Communications Resources

8.7.1 Low-Speed Communication Resource Class

8.7.1.1 Introduction

A part of the host is an interface to a low-speed communications resource class. This will provide bi-directional communications over, for example, a telephone line or cable network return channel. This could be used to support Conditional Access functions, and be used in conjunction with interactive services.

The resource class is defined in a generic fashion so that different underlying communications technologies can be used in a common way. Resource types are defined within the class to support communication to particular types of device using the common object set. The model is a simultaneous bi-directional channel (full duplex) over which communication of arbitrary data is possible. Flow control is applied in both directions between application and host. Data is split into segments for transfer in order to limit the buffer sizes required in both application and host. The flow control protocol also limits the number of buffers required in both application and host.

The APDUs described below are APDUs used by the low-speed communications functions of the host and the application.

8.7.1.2 Requirements

1 Buffers of up to 254 bytes in length shall be accepted in both directions. The application can select a maximum buffer size lower than this if required.

2 Flow control is implemented between host and application. The host shall accept a second buffer for transmission whilst the first is being sent, but the third must wait until the first has been transmitted. A similar protocol is operated for incoming data from the communications interface. It is the host's responsibility to operate flow control on the connections established over the external communications link, and co-ordinated with the application-to-host flow control.

8.7.1.3 Objects Supporting the Low-Speed Communications Resource

Four objects are defined - Comms Cmd, Comms Reply, Comms Send and Comms Rcv. Comms Cmd is sent by the application and allows several management operations on the communications resource to be carried out. Comms Reply is sent by the host and acknowledges a Comms Cmd. It also acknowledges a Comms Send, operating the outgoing flow control to the communications resource. Comms Send is a buffer of data sent to the communications resource. This is data to be sent to line. Comms Rcv is a buffer of data received from the communications resource. This is data received from the line. Buffer size and time-out values previously set by a Comms Cmd object apply to received buffers.

To use a particular communications resource, a session must be created to it using the standard session creation mechanism. Transactions using the comms objects then take place within this session.

8.7.1.4 Comms Cmd

Table 52: Comms Cmd object coding

Syntax	No. of bits	Mnemonic
comms_cmd() {		
comms_cmd_tag	**24**	**uimsbf**
length_field()		
comms_command_id	**8**	**uimsbf**
if (comms_command_id == Connect_on_Channel){		
connection_descriptor()		
retry_count	**8**	**uimsbf**
timeout	**8**	**uimsbf**
}		
if (comms_command_id == Set_Params){		
buffer_size	**8**	**uimsbf**
timeout	**8**	**uimsbf**
}		
if (comms_command_id == Get_Next_Buffer){		
comms_phase_id	**8**	**uimsbf**
}		
}		

Table 53: Connection Descriptor object coding

Syntax	No. of bits	Mnemonic
connection_descriptor() {		
connection_descriptor_tag	**24**	**uimsbf**
length_field()		
connection_descriptor_type	**8**	**uimsbf**
if (connection_descriptor_type == SI_Telephone_Descriptor) {		
telephone_descriptor() /* see DVB/SI specification [4] */		
}		
if (connection_descriptor_type == Cable_Return_Channel_Descriptor) {		
channel_id	**8**	**uimsbf**
}		
}		

comms_command_id	id value
Connect_on_Channel	01
Disconnect_on_Channel	02
Set_Params	03
Enquire_Status	04
Get_Next_Buffer	05
reserved	other values

connection_descriptor_type	type value
SI_Telephone_Descriptor	01
Cable_Return_Channel_Descriptor	02
reserved	other values

Connect_on_Channel establishes communication on the comms resource. The connection_descriptor contains information necessary to establish the connection, for example the telephone number to dial if a modem is used. retry_count allows one or more retries to be attempted, and timeout (in seconds) allows a connection attempt to be aborted if no positive indication of the state of the connection is received within a certain time. A timeout value of zero means wait indefinitely.

Disconnect_on_Channel terminates the connection on the comms resource.

Set Params carries two parameters. The first is a 8-bit value giving the maximum buffer size in bytes, and is limited to a minimum of 1 and a maximum of 254. The second parameter is a 8-bit value giving an input time-out in units of 10ms. If one or more bytes has been received in the current buffer and then a time equal to the timeout has elapsed with no further bytes received, then the buffer is given to the application as a Comms Rcv object. If the buffer fills to the limit set by the buffer_size parameter with no timeout then the buffer is returned immediately.

Enquire_Status has no parameters and generates a Comms Reply object with a parameter indicating the current status of the communications connection.

Get_Next_Buffer operates the flow control protocol on the receive side. After a connection is successfully established on the channel, the application issues Get_Next_Buffer with comms_phase_id set to 0. It must be zero for the first one and the host shall enforce this. If it wishes the application can then also issue Get_Next_Buffer with comms_phase_id set to 1. It can now not issue any more until a buffer of data has been received on channel and sent to the application using the Comms Rcv object. When that happens, it can issue Get_Next_Buffer with the comms_phase_id set to 0 again. comms_phase_id is used with alternating 0,1,0,1... values, and by this means the application controls the rate at which the host can send it data, and the phase value allows the application to unambiguously identify which buffer has been sent. Issuing Get_Next_Buffer elicits an immediate Comms Reply acknowledgement, and the received buffer is sent sometime later using the Comms Rcv object.

8.7.1.5 Comms Reply

Table 54: Comms Reply object coding

Syntax	No. of bits	Mnemonic
comms_reply() {		
comms_reply_tag	**24**	**uimsbf**
length_field()		
comms_reply_id	**8**	**uimsbf**
return_value	**8**	**uimsbf**
}		

comms_reply_id	id value
Connect_Ack	01
Disconnect_Ack	02
Set_Params_Ack	03
Status_Reply	04
Get_Next_Buffer_Ack	05
Send_Ack	06
reserved	other values

In general, positive return values are OK, and negative return values are errors. Zero is the standard OK return values and -1 is the non-specific error. Further error values remain to be defined. The return value for Status_Reply has values 0 = Disconnected and 1 = Connected.

The return values for Send_Ack tell which buffer has been successfully sent. All buffers sent to the comms have a phase, 0 or 1. 0 is the first one sent after a successful 'Connect' Comms Cmd. Subsequent buffers have alternating phases 1,0,1,0 etc. When a 0-phase buffer is acknowledged by a Send_Ack with 0 return value, then the next 0-phase buffer can be sent. Similarly a Send_Ack with a 1 value acknowledges the previous 1-phase buffer, and the next 1-phase buffer can now be sent. This mechanism controls the flow of data from the application to the comms, and requires the host to provide only two buffers - the one currently being sent and the next one.

This object can be sent to the module unsolicited if an error occurs. The only error currently signalled is a disconnection, in which case the host sends comms_reply with comms_reply_id set to Status_Reply and a return value of 0 (denoting 'disconnected').

8.7.1.6 Comms Send

Table 55: Comms Send object coding

Syntax	No. of bits	Mnemonic
comms_send() {		
comms_send_tag	**24**	**uimsbf**
length_field()		
comms_phase_id	**8**	**uimsbf**
for (i=0;i<n;i++){		
message_byte	**8**	**uimsbf**
}		
}		

comms_phase_id takes the values 0 or 1. The first Comms Send after a 'Connect' Comms Cmd must set comms_phase_id to 0. Subsequent Comms Sends will set the values to 1,0,1,0 etc. in an alternating sequence. The host will check this, returning a Comms Reply with a Send_Ack error if the sequence is broken. This allows the application to determine unambiguously which buffer it sent is the one being acknowledged. The complexity of having two phases rather than a simple send-ack-send-ack mechanism is to allow the host to keep the comms continuously fed with data so it can operate at maximum speed.

The maximum number of message bytes is 254.

8.7.1.7 Comms Rcv

Table 56: Comms Rcv object coding

Syntax	No. of bits	Mnemonic
comms_rcv() {		
comms_rcv_tag	**24**	**uimsbf**
length_field()		
comms_phase_id	**8**	**uimsbf**
for (i=0;i<n;i++){		
message_byte	**8**	**uimsbf**
}		
}		

Comms_phase_id indicates which phase of the Get_Next_Buffer cycle to which this data belongs.

8.8 Resource Identifiers and Application Object Tags

8.8.1 Resource Identifiers

Identifiers for the public resources defined in this specification are given here.

Table 57: Resource Identifier values

Resource	class	type	version	resource identifier
Resource Manager	1	1	1	00010041
Application Information	2	1	1	00020041
Conditional Access Support	3	1	1	00030041
Host Control	32	1	1	00200041
Date-Time	36	1	1	00240041
MMI	64	1	1	00400041
Low-Speed Communications	96	see §8.8.1.1	1	0060xxx1
reserved	other values	other values	other values	other values

8.8.1.1 Low-Speed Communications Resource Types

The low-speed communications resource type value encodes two fields. The first field, using bits 0 & 1 of the resource type, is a device number. There may be more than one instance of a particular device, e.g. a modem, and the device number field identifies which one. The second field, using bits 2-9, is the device type proper. This is further split into sub-fields for certain devices, such as modems.

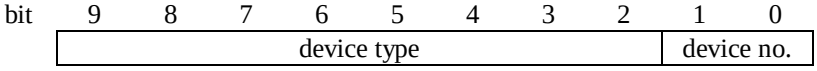

Figure14: Low-Speed Communications resource type structure

The device type field is coded thus:

Description	Value
Modems - see below	00-3F
Serial Ports	40-4F
Cable return channel	50
reserved	51-FF

the resource type field for modems is coded into three fields. Device number is the same as above, but the device type is split into a field coding for the data processing required and a field giving the modem type, coded according to speed specification using the ITU-T V. series numbers.

bit	9	8	7 6 5 4	3 2	1 0
	0	0	modem type	dataproc	device no.

Figure 15: resource type structure for modems

The ‘dataproc’ field is coded thus:

Description	Value
Negotiate	00
No V.42bis	01
No data processing	10
reserved	11

‘Negotiate’ means use both V.42 error correction and V.42bis data compression if it can be negotiated with the far end. ‘No V.42bis’ means negotiate to use V.42 error correction but do not use V.42bis data compression.

The modem type field is coded thus:

Description	Value
reserved	0000
reserved	0001
V.21 (300 bits/sec)	0010
reserved	0011
V.22 (1200bits/sec)	0100
V.22bis (2400 bits/sec)	0101
V.23 (1200/75 bits/sec)	0110
reserved	0111
V.32 (9600/4800 bits/sec)	1000
V.32bis (14.4k bits/sec)	1001
V.34 (28.8k bits/sec)	1010
reserved	1011-1101
V.27ter	1110
V.29	1111

If the particular modem used by the host offers multiple V. series speeds then the host shall offer all the available speeds as different resource types. If one particular resource type is in use then all the others offered by that modem will also become busy.

8.8.2 Application Object Tags

The coding of the apdu_tag follows the ASN.1 rules. Each apdu_tag is coded on three bytes. Among the 24 bits of each apdu_tag, 10 are fixed by the ASN.1 rules as described in the figure below. Only primitive tags are used.

Byte 1 b24 b17	Byte 2 b16 b9	Byte 3 b8 b1
1 0 0 1 1 1 1 1	1 x x x x x x x	0 x x x x x x x

Figure 16: Primitive tag coding

Table 58: Application object tag values

apdu_tag	tag value (hex)	Resource	Direction host<-->app
$T_{profile\_enq}$	9F 80 10	resource mgr.	<--->
$T_{profile}$	9F 80 11	resource mgr.	<--->
$T_{profile\_change}$	9F 80 12	resource mgr.	<--->
$T_{application\_info\_enq}$	9F 80 20	application info.	--->
$T_{application\_info}$	9F 80 21	application info.	<---
T_{enter_menu}	9F 80 22	application info.	--->
$T_{ca\_info\_enq}$	9F 80 30	CA Support	--->
T_{ca_info}	9F 80 31	CA Support	<---
T_{ca_pmt}	9F 80 32	CA Support	--->
$T_{ca\_pmt\_reply}$	9F 80 33	CA Support	<---
T_{tune}	9F 84 00	Host Control	<---
$T_{replace}$	9F 84 01	Host Control	<---
$T_{clear\_replace}$	9F 84 02	Host Control	<---
$T_{ask\_release}$	9F 84 03	Host Control	--->
$T_{date\_time\_enq}$	9F 84 40	Date-time	<---
T_{date_time}	9F 84 41	Date-time	--->
T_{close_mmi}	9F 88 00	MMI	--->
$T_{display\_control}$	9F 88 01	MMI	<---
$T_{display\_reply}$	9F 88 02	MMI	--->
$T_{text\text{-}last}$	9F 88 03	MMI	<---
$T_{text\text{-}more}$	9F 88 04	MMI	<---
$T_{keypad\_control}$	9F 88 05	MMI	<---
$T_{keypress}$	9F 88 06	MMI	--->

Table 58: Application object tag values (end)

apdu_tag	tag value (hex)	Resource	Direction host<-->app
T_{enq}	9F 88 07	MMI	<---
T_{answ}	9F 88 08	MMI	--->
T_{menu_last}	9F 88 09	MMI	<---
T_{menu_more}	9F 88 0A	MMI	<---
T_{menu_answ}	9F 88 0B	MMI	--->
T_{list_last}	9F 88 0C	MMI	<---
T_{list_more}	9F 88 0D	MMI	<---
$T_{subtitle\_segment\_last}$	9F 88 0E	MMI	<---
$T_{subtitle\_segment\_more}$	9F 88 0F	MMI	--->
$T_{display\_message}$	9F 88 10	MMI	<---
$T_{scene\_end\_mark}$	9F 88 11	MMI	<---
T_{scene_done}	9F 88 12	MMI	<---
$T_{scene\_control}$	9F 88 13	MMI	--->
$T_{subtitle\_download\_last}$	9F 88 14	MMI	<---
$T_{subtitle\_download\_more}$	9F 88 15	MMI	--->
$T_{flush\_download}$	9F 88 16	MMI	<---
$T_{download\_reply}$	9F 88 17	MMI	<---
T_{comms_cmd}	9F 8C 00	low-speed comms.	<---
$T_{connection\_descriptor}$	9F 8C 01	low-speed comms.	<---
T_{comms_reply}	9F 8C 02	low-speed comms.	--->
$T_{comms\_send\_last}$	9F 8C 03	low-speed comms.	<---
$T_{comms\_send\_more}$	9F 8C 04	low-speed comms.	<---
$T_{comms\_rcv\_last}$	9F 8C 05	low-speed comms.	--->
$T_{comms\_rcv\_more}$	9F 8C 06	low-speed comms.	--->

Annex A (normative)

PC card-based physical layer

A.1 General description

The physical interface between module and host and the form-factor and mechanical characteristics of the module are a variant of the PC Card specification [6], [7], [8]. Figure A.1 shows a typical module architecture.

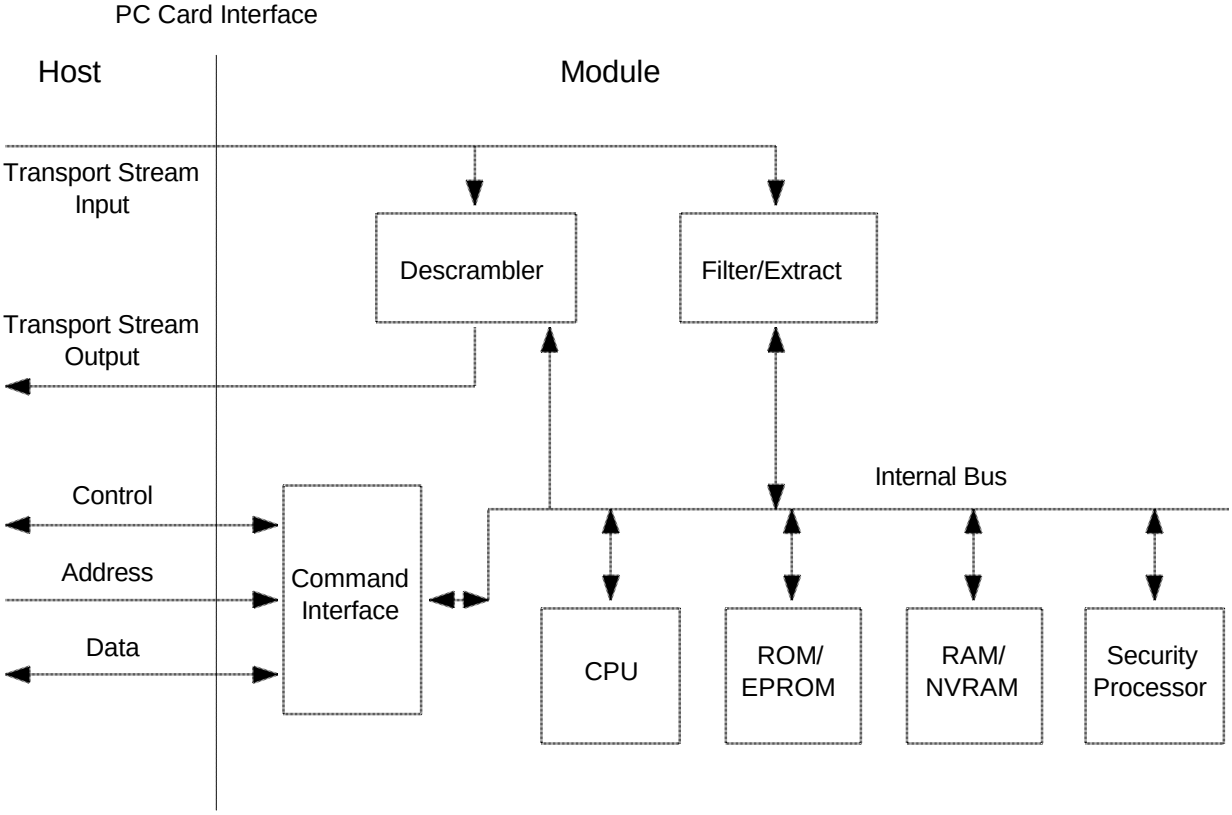

Figure A.1: Typical CA module architecture showing the position of the PC Card interface

A.1.1 PC Card interface

It is described in the following subclauses. It is a variant of the PC Card specification.

The variant provides the following facilities :

- Transport Stream Interface for MPEG-2 data consisting of an 8-bit parallel input to the module and a separate 8-bit parallel output, together with control signals and a byte clock.
- Command Interface for command traffic between host and module consisting of an 8-bit bi-directional data bus together with address and control signals.
- Attribute Memory Interface to allow the host to read the Card Information Structure on the module and to configure the module into its normal operating mode.

A.1.2 Descrambler

The descrambler selectively descrambles transport packets within the Transport Stream. It may also descramble at the Packetised Elementary Stream level if configured to do so. It is configured by particular values (Control Words) loaded into it periodically by the CPU. Since the descrambler may have to descramble several services simultaneously it would normally maintain a list of Control Words associated with the PIDs required to be descrambled. The descrambler will not attempt to descramble any Transport Packets where the transport_scrambling_control flag is set to '00', even if the PID matches with a current control word. This same constraint applies with regard to the PES_scrambling_control flag if PES level scrambling is used.

A.1.3 Filter/Extract

Some of the data required by a CA system to operate successfully is carried within the Transport Stream. The Filter/Extract circuit is configured to extract the data required by the CA module for the programmes/services it has been asked to descramble.

A.1.4 CPU

This runs the CA process and orchestrates the flow of data within the module and between module and host to carry out the CA functions.

A.1.5 ROM/EPROM and RAM/NVRAM

These contain the program and data implementing the CA process. The ROM may also contain the Attribute Memory which is a necessary feature of the PC Card initialisation process.

A.1.6 Security processor

This is a separate processor, manufactured to much higher security requirements than the main CPU, which carries out security functions, such as decryption and also stores secure information, such as keys and entitlements. It may be embedded within the PC Card module or it may reside on an associated detachable module, such as a smart card.

A.2 Electrical interface

The Electrical characteristics of this interface comply with the specified variant of Version 2.1 of the PC Card standard - see A.5.

A.2.1 Transport Stream Interface

MPEG-2 data from the host is presented to the module on an 8-bit data bus MDI0 to MDI7. In addition there are two control signals, MISTRT and MIVAL. These are clocked into the module by a clock signal MCLK. The MPEG-2 data returns from the module on another 8-bit data bus MDO0 to MDO7. Similarly there are two control signals MOSTRT and MOVAL. A detailed description of the operation of this interface is given in A.5.

A.2.2 Command Interface

A.2.2.1. Hardware interface description

The hardware interface consists of several registers occupying 4 bytes of address space on the PC Card interface. Byte offset 0 is the Data Register. This is read to transfer data from the module and written to transfer data to the module. At byte offset 1 are the Control Register and Status Register. Reading at offset 1 reads the Status Register, and writing at offset 1 writes to the Control Register. The Size Register is a 16-bit register at byte offsets 2 and 3. Offset 2 is the Least Significant half and offset 3 the Most Significant half. The register map is shown in figure A.2.1.

Only two address lines, A0 and A1, are decoded by the interface. The host designer is free to place this block of 4 bytes anywhere within his own address space by suitable decoding or mapping of other address lines within the host.

Offset	Register
0	Data Register
1	Command/Status Register
2	Size Register (LS)
3	Size Register (MS)

Figure A.2: Map of Hardware Interface Registers

The Status Register looks like this:

bit	7	6	5	4	3	2	1	0
	DA	FR	R	R	R	R	WE	RE

DA (Data Available) is set to '1' when the module has some data to send to the host.
FR (Free) is set to '1' when the module is free to accept data from the host, and at the conclusion of a Reset cycle initiated by either a module hardware reset, or by the RS command..
R indicates reserved bits. They read as zero.
WE (Write Error) and RE (Read Error) are used to indicate length errors in read or write operations.

The Command Register looks like this:

bit	7	6	5	4	3	2	1	0
	R	R	R	R	RS	SR	SW	HC

RS (Reset) is set to '1' to reset the interface. It does not reset the whole module.
SR (Size Read) is set to '1' to ask the module to provide its maximum buffer size. It is reset to '0' by the host after the data transfer.
SW (Size Write) is set to '1' to tell the module what buffer size to use. It is reset to '0' by the host after the data transfer.
HC (Host Control) is set to '1' by the host before starting a data write sequence. It is reset to '0' by the host after the data transfer.
R indicates reserved bits. They shall always be written as zero.

A.2.2.1.1 Initialisation

During module initialisation, and at other times if there is an error, the host needs to be able to reset the interface. It does this by writing a '1' to the RS bit in the Control Register. The module clears out any data in its data transfer buffer(s) and sets the interface so that it can perform the buffer size negotiation protocol. The module signals that the Reset operation is complete by setting the FR bit to ‘1’.

After initialisation the host must find out the internal buffer size of the module by operating the buffer size negotiation protocol. Neither host nor module may use the interface for transferring data until this protocol has completed. The host starts the negotiation by writing a '1' to the SR bit in the Control Register, waiting for the DA bit to be set and then reading the buffer size by a module to host transfer operation as described below. At the end of the transfer operation the host resets the SR bit to '0'. The data returned will be 2 bytes with the most significant byte first. Modules shall support a minimum buffer size of 16 bytes. The maximum is set by the limitation of the Size Register (65535 bytes). Similarly the host may have a buffer size limitation that it imposes. The host shall support a minimum buffer size of 256 bytes but it can be up to 65535 bytes. After reading the buffer size the module can support, the host takes the lower of its own buffer size and the module buffer size. This will be the buffer size used for all subsequent data transfers between host and module. The host now tells the module to use this buffer size by writing a ‘1’ to the SW bit in the Command Register, waiting until the FR bit is set and then writing the size as 2 bytes of data, most significant byte first, using the host to module transfer operation described below. At the end of the transfer the host sets the SW bit to '0'. The negotiated buffer size applies to both directions of data flow, even in double buffer implementations.

A.2.2.1.2 Host to Module Transfers

The host sets the HC bit and then tests the FR bit. If FR is ‘0’ then the interface is busy and the host must reset HC and wait a period before repeating the test. If FR is ‘1’ then the host writes the number of bytes it wishes to send to the module into the Size register and then writes that number of data bytes to the Data register. This multiple write shall not be interrupted by any other operations on the interface except for reads of the Status Register. When the first byte is written the module sets WE to ‘1’ and sets FR to '0'. During the transfer the WE bit remains at ‘1’ until the last byte is written, at which point it is set to ‘0’. If any further bytes are written then the WE bit is set to ‘1’. At the end of the transfer the host shall reset the HC bit by writing ‘0’ to it.

The host must test the DA bit before initiating the host-to-module cycle above in order to avoid deadlock in the case of a single buffer implementation in the module.

This C code fragment illustrates the host side process:

```
if (Status_Reg & 0x80) /* go to module-to-host transfer (see below) */
Command_Reg = 0x01;
if (Status_Reg & 0x40) {
  Size_Reg[0] = bsize & 0xFF;
  Size_Reg[1] = bsize >> 8;
  for (i=0; i<bsize; i++)
    Data_Reg = write_buf[i];
}
Command_Reg = 0x00;
```

A.2.2.1.3 Module to Host Transfers

Periodically the host tests the DA bit in the Status Register. If DA is ‘1’ then the host reads the Size Register to find out how much data is to be transferred. It then reads that number of data bytes from the Data register. This multiple read shall not be interrupted by any other operations on the interface except for reads of the Status Register. When the first byte is read the module sets RE to ‘1’ and sets DA to '0'. During the transfer the RE bit remains at ‘1’ until the last byte is read, at which point it is set to ‘0’. If any further bytes are read then the RE bit is set to ‘1’. This C code illustrates the host side process:

```
if (Status_Reg & 0x80) {
  bsize = Size_Reg[0] | Size_Reg[1] << 8;
  for (i=0; i<bsize; i++)
    read_buf[i] = Data_Reg;
}
```

The bytes of the Size Register can be read or written in either order.

Note that this interface does not support interrupts of the host by the module. The host is expected to test the DA and FR bits in the status register periodically to determine if communication is required. Support for interrupt-driven operation may be included in a future version of the specification.

A.2.2.2 Hardware support in the module

Any implementation which respects the above specification for the host-module interaction is possible. The module could use a single buffer which is used for transactions in both directions, or to speed up the interface a buffer for each direction of flow could be used. By appropriately controlling the interaction of the fill state of the buffer(s), the HC bit and the FR and DA bits, both interface designs can be accommodated.

A.3 Link layer

A.3.1 Transport Stream Interface

There is no Link Layer on the Transport Stream Interface. The data is in the form of consecutive MPEG-2 Transport Packets, possibly with data gaps within and between Transport Packets.

A.3.2 Command Interface

The Link Layer on the Command Interface does two jobs. It fragments Transport Protocol Data Units (TPDU), if necessary, for sending over the limited buffer size of the Physical Layer, and reassembles received fragments. It also fairly multiplexes several Transport Connections onto the one Link Connection. It does this by interleaving fragments from all the Transport Connections which are currently trying to send TPDUs over the link. It assumes that the Physical Layer transfer mechanism is reliable, that is, it keeps the data in the correct order and neither deletes nor repeats any of it.

A Link Connection is established automatically as a consequence of the establishment of the Physical Layer connection, that is, plugging in the module or powering up, reading the Card Information Structure, and configuring the module in the appropriate mode. No further explicit establishment procedure is required. The size of each Link Protocol Data Unit (LPDU) depends on the size that the host and module negotiated using the SR & SW commands on the interface. Each LPDU consists of a two-byte header followed by a fragment of a TPDU, the total size not exceeding the negotiated buffer size. The first byte of the header is the Transport Connection Identifier for that TPDU fragment. The second byte contains a More/Last indicator in its most significant bit. If the bit is set to '1' then at least one more TPDU fragment follows, and if the bit is set to '0' then it indicates this is the last (or only) fragment of the TPDU for that Transport Connection. All other bits in the second byte are reserved and shall be set to zero. This is illustrated in Figure A.3.1.

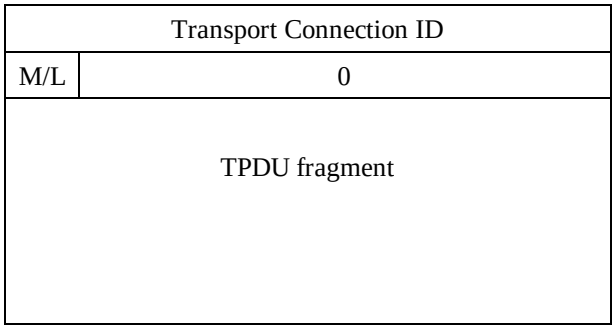

Figure A.3: Layout of Link Protocol Data Unit

Each TPDU shall start in a new LPDU, that is, the LPDU carrying the last fragment of the previous TPDU on a Transport Connection cannot also carry the first fragment of the next one. If more than one Transport Connection currently has TPDUs in transit the Link Layer shall send a fragment for each of them in turn, so that all Transport Connections get a fair apportionment of the communication bandwidth available.

A.4 Implementation-specific Transport sublayer over PC Card Interface

A.4.1 Transport Protocol Objects

The transport protocol on the command interface is a command-response protocol where the host sends a command to the module, using a Command Transport Protocol Data Unit (C_TPDU) and waits for a response from the module with a Response Transport Protocol Data Unit (R_TPDU). The module cannot initiate communication: it must wait for the host to poll it or send it data first. The protocol is supported by eleven Transport Layer objects. Some of them appear only in C_TPDUs from the host, some only in R_TPDUs from the module and some can appear in either. Create_T_C and C_T_C_Reply, create new Transport Connections. Delete_T_C and D_T_C_Reply, clear them down. Request_T_C and New_T_C allow a module to request the host to create a new Transport Connection. T_C_Error allows error conditions to be signalled. T_SB carries status information from module to host. T_RCV requests waiting data from a module and T_Data_More and T_Data_Last convey data from higher layers between host and module. T_Data_Last with an empty data field is used by the host to poll regularly for data from the module when it has nothing to send itself. In all objects there is a tag_field and a length_field coded according to the rules defined in [3], and a t_c_id field which is a single octet. The length_field coding is shown in figure 6.

A.4.1.1 Command TPDU

The Command TPDU (C_TPDU) conveys Transport Protocol objects from host to module.

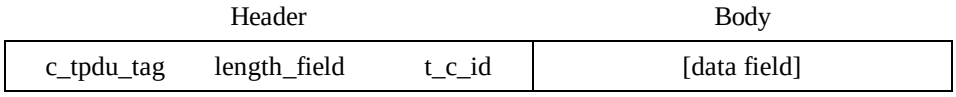

Figure A.4: C_TPDU structure

Table A.1: C_TPDU coding

Syntax	No. of bits	Mnemonic
C_TPDU() {		
c_tpdu_tag	**8**	**uimsbf**
length_field()		
t_c_id	**8**	**uimsbf**
for (i=0;i<length_value-1;i++) {		
data_byte	**8**	**uimsbf**
}		
}		

The C_TPDU is made of two parts :

- a mandatory header made of a Tag value c_tpdu_tag, coding the TPDU command, a length_field, coding the length of all the following fields, and a transport connection identifier noted t_c_id.
- a conditional body of variable length equal to the length coded by length_field minus one.

A.4.1.2 Response TPDU

The Response TPDU (R_TPDU) conveys Transport Protocol objects from module to host.

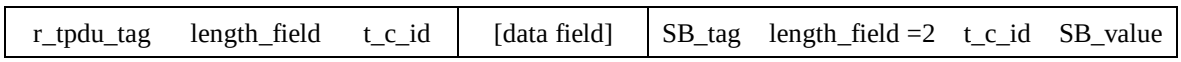

Figure A.5: R_TPDU structure

Table A.2: R_TPDU coding

Syntax	No. of bits	Mnemonic
R_TPDU() {		
r_tpdu_tag	**8**	**uimsbf**
length_field()		
t_c_id	**8**	**uimsbf**
for (i=0;i<length_value-1;i++) {		
data_byte	**8**	**uimsbf**
}		
SB_tag	**8**	**uimsbf**
length_field()=2		
t_c_id	**8**	**uimsbf**
SB_value	**8**	**uimsbf**
}		

The R_TPDU is made of three parts :

- a conditional header made of a Tag value r_tpdu_tag, coding the TPDU response, a length field, coding the length of the following transport connection identifier and data fields, and a transport connection identifier field noted t_c_id. The status is not included in the calculation of length_field.
- a conditional body of variable length equal to the length coded by length_field minus one.
- a mandatory Status made of a Status tag SB_tag, a length_field equal to 2, a transport connection identifier and a one-byte Status Byte value (SB_value) coded according to figure A.6 and table A.3 below.

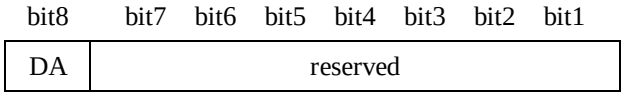

Figure A.6: SB_value coding

Table A.3 - coding of bit8 of SB_value

The 1-bit DA (Data Available) indicator field indicates whether the module has a message available in its output buffer for the host. The host has to issue a Receive_data C_TPDU to get the message (see A.4.1.11.2). The coding of DA indicator is given in table A.3. The 'reserved' field shall be set to zero.

bit8	meaning
0	no message available
1	message available

A.4.1.3 Chaining of command or response TPDU data fields

The data field which is in a C_TPDU or in a R_TPDU may be split into several blocks of smaller sizes if required by the transmission and reception buffer sizes of the host and of the module.

The chaining is performed by using two different c_TPDU_tag values (M_c_TPDU_tag and L_c_TPDU_tag) in the case of C_TPDU, or two different r_TPDU_tag values (M_r_TPDU_tag and L_r_TPDU_tag) in the case of R_TPDU. All blocks of the data field, except the last one, is sent within a C_TPDU (or R_TPDU) with a M_c_TPDU_tag (or M_r_TPDU_tag) value. This tag value indicates that more data will be sent in another C_TPDU (or R_TPDU). The last block of data field is sent within a C_TPDU (or R_TPDU) with a L_c_TPDU_tag (or L_r_TPDU_tag) value. When the last block is received, the receiving entity concatenates all the received data fields.

This mechanism is valid for all C_TPDUs (or R_TPDUs) with two defined tag values. It is illustrated in figures A.7 and A.8 below.

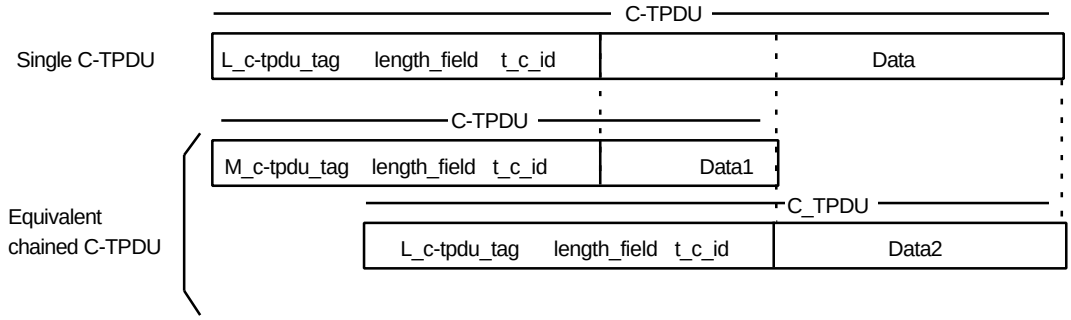

Figure A.7 - illustration of the chaining mechanism with C_TPDU

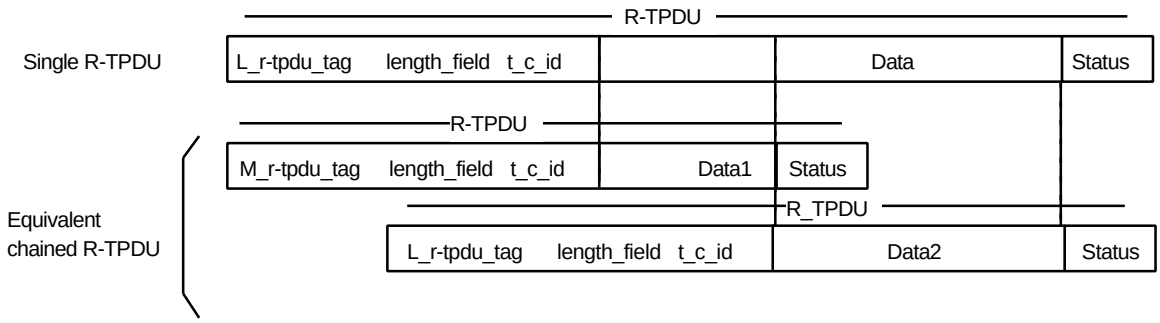

Figure A.8: Illustration of the chaining mechanism with R_TPDU

A.4.1.4. Create Transport Connection (Create_T_C)

Header

create_T_C_tag	length_field	t_c_id

Figure A.9: Create_T_C structure

Table A.4: Create_T_C coding

Syntax	No. of bits	Mnemonic
Create_T_C() {		
create_T_C_tag	**8**	**uimsbf**
length_field()=1		
t_c_id	**8**	**uimsbf**
}		

The Create_T_C object is made of only one part :

- a mandatory header made of a Tag value create_T_C_tag, coding the Create_T_C object, a length_field equal to one, and a transport connection identifier noted t_c_id.

A.4.1.5 Create Transport Connection Reply (C_T_C_Reply)

Header

C_T_C_Reply_tag	length_field	t_c_id

Figure A.10: C_T_C_Reply structure

Table A.4: C_T_C_Reply coding

Syntax	No. of bits	Mnemonic
C_T_C_Reply() {		
C_T_C_Reply_tag	**8**	**uimsbf**
length_field() = 1		
t_c_id	**8**	**uimsbf**
}		

The C_T_C_Reply object is made of only one part :

- a mandatory header made of a Tag value C_T_C_Reply_tag, coding the C_T_C_Reply object, a length_field equal to one, and a transport connection identifier.

A.4.1.6. Delete Transport Connection (Delete_T_C)

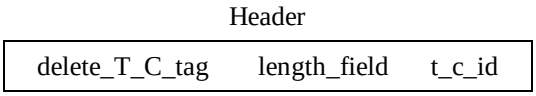

Figure A.10: Delete_T_C structure

Table A.6: Delete_T_C coding

Syntax	No. of bits	Mnemonic
Delete_T_C() {		
delete_T_C_tag	**8**	**uimsbf**
length_field() = 1		
t_c_id	**8**	**uimsbf**
}		

The Delete_T_C object is made of only one part :

- a mandatory header made of a Tag value delete_T_C_tag, coding the Delete_T_C object, a length_field equal to one, and a transport connection identifier.

A.4.1.7 Delete Transport Connection Reply (D_T_C_Reply)

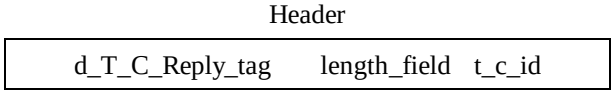

Figure A.12: D_T_C_Reply structure

Table A.7 - D_T_C_Reply coding

Syntax	No. of bits	Mnemonic
D_T_C_Reply() {		
d_T_C_Reply_tag	**8**	**uimsbf**
length_field() = 1		
t_c_id	**8**	**uimsbf**
}		

The D_T_C_Reply object is made of only one part :

- a mandatory header made of a Tag value d_T_C_Reply_tag, coding the D_T_C_Reply object, a length_field equal to one, and a transport connection identifier.

A.4.1.8 Request Transport Connection (Request_T_C)

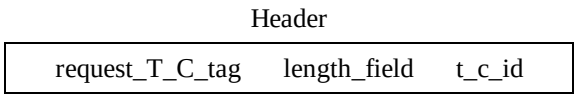

Figure A.13: Request_T_C structure

Table A.8: Request_T_C coding

Syntax	No. of bits	Mnemonic
Request_T_C() {		
request_T_C_tag	**8**	**uimsbf**
length_field() = 1		
t_c_id	**8**	**uimsbf**
}		

The Request_T_C object is made of only one part :

- a mandatory header made of a Tag value request_T_C_tag, coding the Request_T_C object, a length_field equal to one, and a transport connection identifier.

A.4.1.9 New Transport Connection (New_T_C)

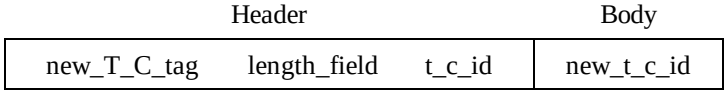

Figure A.14: New_T_C structure

Table A.9: New_T_C coding

Syntax	No. of bits	Mnemonic
New_T_C() {		
new_T_C_tag	**8**	**uimsbf**
length_field() = 2		
t_c_id	**8**	**uimsbf**
new_t_c_id	**8**	**uimsbf**
}		

The New_T_C object is made of two parts :

- a mandatory header made of a Tag value new_T_C_tag, coding the New_T_C object, a length_field equal to two and a transport connection identifier.
- a mandatory body consisting of the transport connection identifier for the new connection to be established.

A.4.1.10 Transport Connection Error (T_C_Error)

Header			Body
T_C_Error_tag	length_field	t_c_id	error_code

Figure A.15: T_C_Error structure

Table A.10 T_C_Error coding

Syntax	No. of bits	Mnemonic
T_C_Error() {		
T_C_Error_tag	**8**	**uimsbf**
length_field() = 2		
t_c_id	**8**	**uimsbf**
error_code	**8**	**uimsbf**
}		

The T_C_Error object is made of two parts :

- a mandatory header made of a Tag value T_C_Error_tag, coding the T_C_Error object, a length_field equal to two and a transport connection identifier.
- a mandatory body consisting of the error code for the particular error being signalled.

Error codes used are as follows:

Table A.11: Error code values

error code	meaning
1	no transport connections available

A.4.1.11 List of C_TPDUs and associated R_TPDUs

A.4.1.11.1 Send Data command

This command is used by the host, when the transport connection is open, either to send data to the module or to get information given by the status byte. The module replies with the status byte.

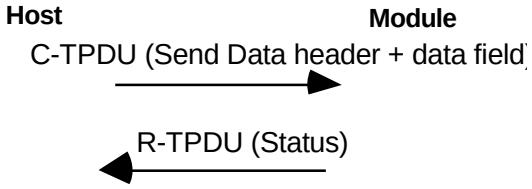

Figure A.16: Send Data command/response pair

Table A.12: Coding of Send Data C_TPDU

c_TPDU_tag	M_c_TPDU_tag : T_{data_more} L_c_TPDU_tag : T_{data_last}
length_field	length of data field according to [3]
t_c_id	transport connection identifier
data field	subset of (TLV TLV ... TLV)

Table A.13: Coding of Send Data R_TPDU

Status	according to figure A.6

A SEND DATA C_TPDU with L=1 (no data field) can be issued by the host just to get information given by the status byte (see polling function, subclause A.4.1.12).

A.4.1.11.2 Receive Data command

This command is used by the host to receive data from the module.

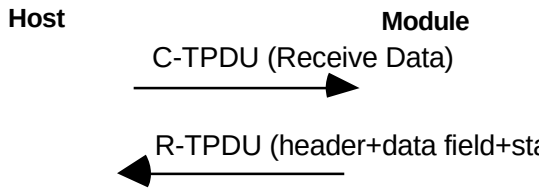

Figure A.17: Receive Data command/response pair

Table A.14: Ccoding of Receive Data C_TPDU

c_TPDU_tag	T_{RCV}
length_field	c_TPDU_length is set to '1'
t_c_id	transport connection identifier

Table A.15: Coding of Receive Data R_TPDU

r_TPDU_tag	M_r_TPDU_tag : T_{data_more} L_r_TPDU_tag : T_{data_last}
length_field	length of data field according to [3]
t_c_id	transport connection identifier
data field	subset of (TLV TLV ... TLV)
Status	according to figure A.6

A.4.1.12 Rules for the polling function

The polling function consists in sending a command to the module in order to know whether it has data to send to the host or not.
This function is provided by the host which has regularly to issue a SEND DATA C_TPDU with length L equal to 1 (t_c_id field only, no data field). As long as data is not available, the module replies with the status byte having the DA indicator set to 0. When data is available, the module replies with the status byte having DA indicator set to 1. The host can then issue one or several RECEIVE DATA C_TPDU, that will provoke transmission of data, until the DA indicator of the status byte is set to 0 again. Whilst the module still has data to send, that is, the host has received status bytes with the DA indicator set, then the poll function shall be suspended. It shall be restarted when a status byte is received with the DA indicator not set.

The maximum period of the polling function is equal to 100ms.

At each poll a timeout of 300ms is started, and is reset when the poll response is received. If no poll response has been received within that time, then the transport connection is deleted by the host in the normal way. The host does not send any additional polls whilst waiting for a poll response, even if its normal poll interval is exceeded.

A.4.1.13 List of transport tags

The coding of the tpdu_tag follows the ASN.1 rules. Each tpdu_tag is coded in one byte.

Table A.16: Coding of Transport tags

tpdu_tag	tag value (hex)	Primitive or Constructed	Direction host <-->module
T_{SB}	'80'	P	<----
T_{RCV}	'81'	P	---->
$T_{create\_t\_c}$	'82'	P	---->
$T_{c\_t\_c\_reply}$	'83'	P	<----
$T_{delete\_t\_c}$	'84'	P	<---->
$T_{d\_t\_c\_reply}$	'85'	P	<---->
$T_{request\_t\_c}$	'86'	P	<----
$T_{new\_t\_c}$	'87'	P	---->
$T_{t\_c\_error}$	'88'	P	---->
T_{data_last}	'A0'	C	<---->
T_{data_more}	'A1'	C	<---->

A.5 PC Card subset to be used by conformant Hosts and Modules

A.5.1 Objectives

The objective of this clause is to capture all the necessary information required to interpret the PC Card specification [6], [7], [8] and make the appropriate choices to successfully develop a minimally conformant host and module.

A.5.2 Introduction

This clause defines the minimum subset of the PC Card specification which is to be used by both hosts and modules. Nothing in this clause prevents hosts from implementing more than the minimum subset, for example to support other types of PC Card devices. However, all modules shall be implemented such that they will operate correctly in hosts conforming to the minimum subset specification described here.

This clause makes extensive reference to the PC Card specification documents [6], [7], [8].

A.5.3 Terminology

In several places in this clause hexadecimal numbers are shown. The format used is the hexadecimal number, using upper case A to F for the final 6 digits, followed by the lower case 'h' - for example 7FFh.

A.5.4 Physical Specification

A.5.4.1 Card dimensions

The host shall accept both Type 1 and Type II PC Cards, cf. clause 3 in [7]. Host support for Type III cards is optional.

A.5.4.2 Connector

Clause 4 in [7] applies.

A.5.4.3 PC Card Guidance

Clause 5 in [7] applies.

A.5.4.4 Grounding/EMI Clips

Clause 6 in [7] applies.

A.5.4.5 Connector Reliability

Clause 7 in [7] applies.

A.5.4.6 Connector Durability

Subclause 8.2 (Harsh Environment) in [7] applies.

A.5.4.7 PC Card Environmental

Clause 9 in [7] applies. With regard to temperature specification the module shall operate at up to 55 deg Celsius, as defined in the PC Card Specification. This is primarily to enable reliable battery operation in modules. In order to facilitate this operating environment limit the host shall limit the temperature rise between the ambient environment outside the host and the ambient environment surrounding the module to 15 deg Celsius when the module is dissipating its full rated power. Note that this may not guarantee that the 55 deg Celsius limit for modules is met in all environmental conditions otherwise acceptable to the host. Module designers shall minimise the risk of misoperation of modules containing batteries by following the recommendations in the PC Card standard on battery placement and host designers should be aware of these recommendations when doing host thermal design. Note that designers will also have to conform to any relevant mandatory safety specifications with regard to module temperature.

A.5.5 Electrical Specification

A Common Interface module is implemented as a variant of the 16-bit PC Card Electrical Interface (Clause 4 of [6]). The command interface uses the least significant byte of the data bus, together with the lower part of the address bus (A0-A14), and appropriate control signals. The command interface operates in I/O interface mode. The upper address lines (A15-A25), the most significant half of the data bus (D8-D15), and certain other control signals are redefined for the this interface variant.

When first plugged in and before configuration, a module conforming to this interface specification shall behave as a Memory-Only device with the following restrictions:

a) Signals D8-D15 shall remain in the high-impedance state.

b) 16-bit read and write modes are not available. CE2# shall be ignored and interpreted by the module as always being in the ‘High’ state.

c) Address lines A15-A25 shall not be available for use as address lines. The maximum address space available on the module is limited to 32768 bytes (16384 bytes of Attribute Memory as it only appears at even addresses).

d) Signals BVD1 and BVD2 shall remain ‘High’.

A.5.5.1 Card Type Detection

Clause 3 of [6] applies. Hosts shall support 5v working and may optionally support 3.3v working. In the latter case hosts shall use the detection mechanisms described in [6] to determine module operating voltage. Hosts shall follow the rules for type of socket according to whether they provide 3.3v working - Subclause 3.2 of [6]. Hosts need not support the detection mechanisms for CardBus PC Cards, but may optionally do so - Subclause 3.3 of [6].

A.5.5.2 Pin assignments

When in memory card mode, just after reset, the pin assignments in the left hand column of Tables 4-1 and 4-2 of [6] are used. When the module is configured as the Common Interface variant during the initialisation process, the following reassignments are made: The pins carrying signals A15-A25, D8-D15, BVD1, BVD2 and VS2# are used to provide high-speed input and output buses for the MPEG-2 multiplex data. All other pins retain their assignment as an I/O & Memory Card interface, except that IOIS16# is never asserted and CE2# is ignored.

The pin assignments for the custom interface are defined in table A.17 below.

Table A.17: Pin assignments for this PC CARD variant

Pin	Signal	I/O	Function
1	GND		Ground
2	D3	I/O	Data bit 3
3	D4	I/O	Data bit 4
4	D5	I/O	Data bit 5
5	D6	I/O	Data bit 6
6	D7	I/O	Data bit 7
7	CE1#	I	Card enable 1
8	A10	I	Address bit 10
9	OE#	I	Output enable
10	A11	I	Address bit 11
11	A9	I	Address bit 9
12	A8	I	Address bit 8
13	A13	I	Address bit 13
14	A14	I	Address bit 14
15	WE#	I	Write enable
16	IREQ#	O	Interrupt request
17	VCC		Vcc
18	VPP1		Program voltage 1
19	MIVAL	I	MP in valid
20	MCLKI	I	MPEG-2 Clock input
21	A12	I	Address bit 12
22	A7	I	Address bit 7
23	A6	I	Address bit 6
24	A5	I	Address bit 5
25	A4	I	Address bit 4
26	A3	I	Address bit 3
27	A2	I	Address bit 2
28	A1	I	Address bit 1
29	A0	I	Address bit 0
30	D0	I/O	Data bit 0
31	D1	I/O	Data bit 1
32	D2	I/O	Data bit 2
33	IOIS16#		16bit I/O (always high)
34	GND		Ground

Pin	Signal	I/O	Function
35	GND		Ground
36	CD1#	O	Card detect 1
37	MDO3	O	MP data out 3
38	MDO4	O	MP data out 4
39	MDO5	O	MP data out 5
40	MDO6	O	MP data out 6
41	MDO7	O	MP data out 7
42	CE2#	I	Card enable 2
43	VS1#	O	Voltage sense 1
44	IORD#	I	I/O read
45	IOWR#	I	I/O write
46	MISTRT	I	MP in start
47	MDI0	I	MP data in 0
48	MDI1	I	MP data in 1
49	MDI2	I	MP data in 2
50	MDI3	I	MP data in 3
51	VCC		Vcc
52	VPP2		Program voltage 2
53	MDI4	I	MP data in 4
54	MDI5	I	MP data in 5
55	MDI6	I	MP data in 6
56	MDI7	I	MP data in 7
57	MCLKO	O	MPEG-2 Clock output
58	RESET	I	Card reset
59	WAIT#	O	Extend bus cycle
60	INPACK#	O	Input port ack
61	REG#	I	Register select
62	MOVAL	O	MP out valid
63	MOSTRT	O	MP out start
64	MDO0	O	MP data out 0
65	MDO1	O	MP data out 1
66	MDO2	O	MP data out 2
67	CD2#	O	Card detect 2
68	GND		Ground

A.5.5.3 16-bit PC Card features

Subclause 4.3 in [6] applies. It is recommended that at least 12 bits of address be decoded on the module (4096 bytes) during memory accesses. A lower number of bits, as specified in the CIS, shall be decoded during I/O accesses.

A.5.5.4 Signal Description

Subclauses 4.4, 4.6 and 4.7 of [6] apply. The following information also amends the specification:

MPEG-2 transport stream interface input and output buses are provided (MDI0-7, MDO0-7). Control signals MCLKI, MCLKO, MISTRT, MIVAL, MOSTRT, MOVAL are also provided.

MCLKI runs at the rate at which bytes are offered to the module on MDI0-7. MCLKO runs at the rate at which bytes are offered by the module on MDO0-7. For modules which pass the transport stream through, then MCLKO will in most cases be a buffered version of MCLKI with a small delay. For modules which originate data, for example a detachable front-end or a network connection, then MCLKO may be derived from that data source. Figure A.18 shows the relative timing relationship of the data signals associated with the MPEG-2

transport stream interface and MCLKI, MCLKO, and Table A.18 gives limits to these timing relationships. Note that the specification for output timing limits can normally be met easily by generating the output from the falling edge of MCLKO. There is no specification for delay between MCLKI and MCLKO. In the case of a module providing its own MCLKO, they may not even be the same frequency. Hosts shall daisy-chain MCLKO from one module to MCLKI of the next module where they support multiple Common Interface sockets, and the transport stream clock reference for the rest of the host shall be derived from MCLKO of the final socket in the chain. Both MCLKI and MCLKO shall be continuous. It is not intended that burst clocking should be employed. Bursty data is handled by the appropriate use of MIVAL and MOVAL.

MISTRT is valid during the first byte of each transport packet on MDI0-7. The edge timing of this signal shall be the same as for MDI0-7.

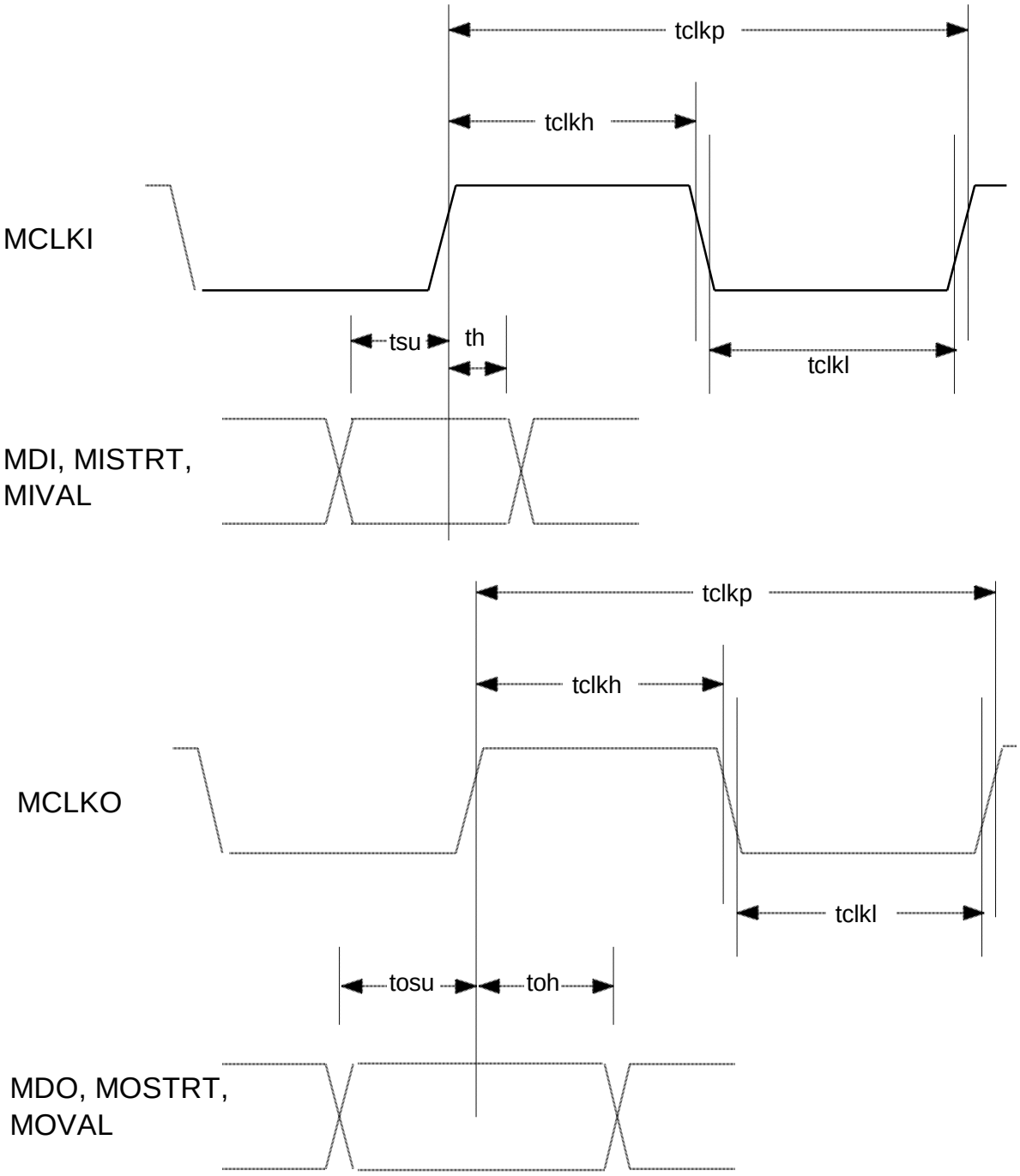

Figure A.18: Timing relationships for Transport Stream Interface signals

Table A.18: Timing relationship limits

Item	Symbol	Min	Max
Clock period	tclkp	111 ns	
Clock High time	tclkh	40 ns	
Clock Low time	tclkl	40 ns	
Input Data Setup	tsu	15 ns	
Input Data Hold	th	10 ns	
Output Data Setup	tosu	20 ns	
Output Data Hold	toh	15 ns	

MIVAL indicates valid data bytes on MDI0-7. All bytes of a transport packet may be consecutive in time, in which case MIVAL will be at logic 1 for the whole of the duration of the transport packet. However certain clocking strategies adopted in hosts may require there to be gaps of one or more byte times between some consecutive bytes within and/or between transport packets. In this case MIVAL will go to logic 0 for one or more byte times to indicate data bytes which should be ignored.

MDO0-7 is the output bus for the MPEG-2 transport stream interface. Where MCLKO is derived from MCLKI, and correspondingly the data on MDO0-7 is a delayed and possibly descrambled version of the data on MDI0-7, then the timing relationship between the input data and the output data shall be governed by the rules in 5.4.2 of this specification.

MOSTRT is valid during the first byte of an output transport packet.

MOVAL indicates the validity of bytes on MDO0-7 in a similar manner to MIVAL. MOVAL may not necessarily be a time-delayed version of MIVAL (see 5.4.2 of this specification).

Support for Interrupt Requests by hosts as defined in subclause 4.4.7 of [6] is optional. This function shall not be used by CA modules. Interrupt support by modules may be added to a future version of the specification and host designers may wish to note this. Whenever the module responds to an I/O operation it shall assert INPACK# (see subclause 4.4.22 of [6]).

A.5.5.5 Memory function

Subclause 4.6 in [6] applies. Attribute Memory function support by hosts is mandatory. Note that Attribute Memory is byte-wide - Attribute Memory data only appears on data lines D7-D0. Also consecutive bytes are at consecutive *even* addresses (0, 2, 4, etc.). Note also that Attribute memory must still be able to be read or written to even when the module is configured to operate with the DVB Common Interface. Common Memory function support in the host is optional. Modules shall not use Common Memory.

A.5.5.6 Timing Functions

Subclause 4.7 in [6] applies. Attribute Memory support by hosts is mandatory. Common Memory support in the host is optional.

A.5.5.7 Electrical interface

Subclause 4.9 in [6] applies. Support by hosts for overlapping I/O address windows as defined in subclause 4.9.3.2 is optional. Modules shall use an independent I/O address window 4 bytes in size.

A.5.5.8 Card detect

Subclause 4.10 in [6] applies.

A.5.5.9 Battery Voltage Detect

Modules shall not implement or require this function. Support for it by hosts is optional.

A.5.5.10 Power-up and power-down

Subclause 4.12 in [6] applies, including the average current during configuration specification. The module shall specify its operating current in the CIS (subclause 3.3.2 of [8]). Each module shall neither consume nor dissipate more than 1.5 watts. Additionally the power supply current to each module (sum of Vcc current and Vpp current) shall not exceed 300mA long term, and the short-term peak current is limited to 500mA for no more than 1ms. Hosts need not support alternate values of Vpp. If they do not it shall be the same voltage as Vcc.

A.5.5.11 I/O Function

Subclause 4.13 in [6] applies excepting that modules shall only use 8-bit read and write modes.

A.5.5.12 Function Configuration

Subclause 4.14 in [6] applies. Modules shall support only the Configuration Option Register. Host support for registers other than the Configuration Option Register is optional.

A.5.5.13 Card Configuration

Subclauses 4.15 and 4.15.1 in [6] apply.

A.5.6 Metaformat Specification

This subclause defines the minimum set of tuples that a Common Interface module must have in its Card Information Structure. This is also the minimum set that a host needs to be able to recognise.

Only clauses 1, 2 and 3 of [8] apply.

The following tuples are the minimum set used by Common Interface modules and are the only set required to be recognised by hosts. The tuple chain formed by them must be the first one in Attribute Memory, in this order, though it may be followed by others, not required to be recognised by hosts. The host must take account of the possibility that these may be followed by other tuples. The chain shall be terminated by a CISTPL_NOLINK tuple. All subclause references here are to subclauses in [8].

1 CISTPL_DEVICE_OA: Coded according to subclauses 3.2.3 & 3.2.4.

2 CISTPL_DEVICE_OC: Coded according to subclauses 3.2.3 & 3.2.4.

3 CISTPL_VERS_1: TPLLV1_MAJOR = 05h; TPLLV1_MINOR = 00h; other fields defined by module manufacturer according to subclause 3.2.10.

4 CISTPL_MANFID: Defined by manufacturer according to subclause 3.2.9.

5 CISTPL_CONFIG: Defined by manufacturer according to subclause 3.3.4. The Configuration Register Base Address shall not be greater than FFEh. There shall be one subtuple in the TPCC_SBTPL field - see subclause 3.3.4.5. In this subtuple STCI_IFN shall contain the ID number 0241h issued by PCMCIA for this class of module, and STCI_STR shall contain the single string 'DVB_CI_V1.00'. The value of STCI_IFN is the specific indication that this is a DVB Common Interface compliant module. The STCI_STR string indicates the version number. The last 5 characters of this string will always be 'Vx.xx' where x is a digit, and will indicate the specification version with which the module complies.

6 CISTPL_CFTABLE_ENTRY: Defined according to subclause 3.3.2. Modules shall support at least one of these. For the first entry TPCE_INDX has both bits 6 (Default) and 7 (Intface) set. The Configuration Entry Number can be any convenient value other than zero. TPCE_IF = 04h - indicating Custom Interface 0. TPCE_FS shall indicate the presence of both I/O and power configuration entries. TPCE_IO is a 1-byte field with the value 22h - see subclause 3.3.2.6. The information means: 2 address lines are decoded by the module and it uses only 8-bit accesses. The power configuration entry - required by subclause A.5.5.10 of this specification, shall follow the PC Card specification, [8]. The CFtable entry also contains the two following subtuples:

7 STCE_EV: see subclause 3.3.2.10.1. Only the system name 'DVB_HOST' is present.

8 STCE_PD: see subclause 3.3.2.10.2. Only the physical device name 'DVB_CI_MODULE' is present.

9 CISTPL_END: the value FFh. If the CA module contains other tuples in addition to those defined above then these will come before CISTPL_END.

Annex B (informative)

Additional objects

B.1 Authentication

This optional resource is included in hosts which support authentication to enable the transmission of authentication commands between a detachable module and a host such that the use of any authentication procedure is under the control of a CA system operator only in respect of signals controlled by him. A broadcaster and/or CA system operator who does not wish to use the authentication protocol must be able to transmit signals which will pass through the common interface without the intervention of any authentication process.

The resource consists of two objects, Authentication Request and Authentication Response which are used to implement an authentication protocol. The particular use is defined between CA manufacturers and manufacturers of hosts which include this resource and is not defined here.

B.1.1 Authentication Request and Authentication Response

These objects are identical except for the tag value.

Syntax	No. of bits	Mnemonic
auth_req () {		
auth_req_tag	**24**	**uimsbf**
length_field()		
auth_protocol_id	**16**	**uimsbf**
for (i=0; i<n; i++) {		
auth_req_byte	**8**	**uimsbf**
}		
}		

Syntax	No. of bits	Mnemonic
auth_resp () {		
auth_resp_tag	**24**	**uimsbf**
length_field()		
auth_protocol_id	**16**	**uimsbf**
for (i=0; i<n; i++) {		
auth_resp_byte	**8**	**uimsbf**
}		
}		

B.1.2 Authentication resource coding

Resource	class	type	version	resource identifier
Authentication	16	1	1	00100041

apdu_tag	tag value (hex)	Resource	Direction host<-->app
T_{auth_req}	9F 82 00	Authentication	<---
T_{auth_resp}	9F 82 01	Authentication	--->

B.2 EBU Teletext Display Resource

The resource defines an object and protocol for communication with the host if it signals that the EBU Teletext (EBT) Display resource is present.
EBT is a means of delivering text-oriented data for display on a television screen. It is normally transported on spare lines in the field blanking interval of conventional TV signals, and will be carried in data transport streams in the multiplex of MPEG-2 broadcasts. EBT defines a particular display capability in the TV set, and the following set of objects is defined to convey data for display using this capability. The full specification is in [B1] but the salient features for display purposes are repeated here.

B.2.1 Display characteristics

The display consists of a two-dimensional alpha-mosaic character grid of from 40 to 160 characters per row and up to 101 rows. However on 4:3 aspect ratio 625-line displays this will be limited to 40 characters per row and 25 rows. The specification uses a defined character set for display but it also allows Dynamically Redefinable Character Sets (DRCS). It uses a standard set of colours for display but also allows a map of different colour tables to be downloaded. As well as an alpha-mosaic display capability, the standard also provides for alpha-geometric and alpha-photographic displays but the current version of this resource does not support them. The display shall accept Presentation Level Two syntax.

B.2.2 Object communication philosophy

The communications philosophy adopted here is to provide the EBT display with data as if it had been delivered over air on a TV line or in a MPEG-2 transport stream. A EBT Packets object is provided for this purpose. A session is set up to the EBT display resource, which succeeds if no other application is using it. Whilst the application has control any incoming teletext service is denied access to the display. The display is allocated on a first come first served basis to the applications. After the first EBT Packets object is received the display shall switch from picture to text mode and display what has been loaded. Further EBT Packets objects can be sent if required. On clear-down of the session the display shall revert to picture mode.

B.2.3 EBT Packets

Syntax	No. of bits	Mnemonic
ebt_packets() {		
ebt_packet_tag	**24**	**uimsbf**
length_field()		
for (i:=0; i<no_of_packets; i++) {		
for (j=0; j<42; j++) {		
ebt_packet_byte	**8**	**uimsbf**
}		
}		
}		

The data in a EBT Packets object is several rows, each of which is 42 bytes long, matching the format for 625-line applications in the EBT specification. In each row the first two bytes contain the Magazine and Packet Address fields. Each byte is Hamming coded with 4 bits of information and 4 parity bits. Of the eight information bits in the two bytes the first three bits form the Magazine number (X) and the last 5 bits form the Packet Address (Y). By convention X is in the range 1-8 where 8 is represented by 000. Y is in the range 0-31.

The coding of the other 40 bytes depends upon the value of the X and Y fields. In general Y=0 packets are header packets and contain further Hamming coded information, followed by display text. Y=1 to 25 packets contain 40 bytes of text and display control characters intended for display on the screen. Y=26 to 31 packets in general carry ancillary display and control information to provide enhanced features.

B.2.4 EBU Teletext resource coding

Resource	class	type	version	resource identifier
EBU Teletext	128	1	1	00800041

apdu_tag	tag value (hex)	Resource	Direction host<-->app
$T_{ebt\_packets}$	9F 90 00	EBU Teletext	<---

B.2.5 References

[B1] EBU SPB 492: Teletext Specification (625 line television systems), EBU, Geneva.

B.3 Smart Card Reader Resource Class

This document describes an optional smart card reader resource which can be provided by a host directly or by another module. This resource uses commands at the card instruction level (header, data and status bytes). The commands are defined in the ISO 7816-1,2,3 specifications. Four objects are defined. Smart Card Cmd and Smart Card Reply perform control and response functions. Smart Card Send and Smart Card Rcv send and receive data. The resource allows multiple sessions to be set up to it so that different applications can interrogate the resource, but only one session can be in the 'connected' state at any time.

The resource is intended to support a smart card reader in a host or on a module used for short-term sessions, such as Bank Card, teleshopping or pay-per-view transactions. It is not recommended that it be used for long-period sessions to smart cards required for ongoing security operations to maintain access to a scrambled service. This is because the smart card reader resource is thereby tied up and not available for other uses, and also because no guarantee of bounded response delay can be given.

B.3.1 Objects

B.3.1.1 Smart Card Cmd

Syntax	No. of Bits	Mnemonic
smart_card_cmd () {		
smart_card_cmd_tag	**24**	**uimsbf**
length_field ()		
smart_card_cmd_id	**8**	**uimsbf**
}		

smart_card_cmd_id	id value
connect	01
disconnect	02
power_on_card	03
power_off_card	04
reset_card	05
read_status	06
read_answ_to_reset	07
reserved	other values

This command is issued by the application once a session has been set up to the resource. In response a Smart Card Reply object is returned by the resource. The following commands are available:

- **connect**: connect this session to the card reader. If the connection is successful then the reply id will be 'connected', with an appropriate card status - card_inserted, no_card, etc. If the card reader is already connected on another session then the reply id 'busy' is returned.

- **disconnect**: disconnect the card reader from this session. The card is powered off. The reply id 'free' is returned with the appropriate card status. If the card reader is connected in another session then the reply id 'busy' is returned.

- **power_on_card**: switch on the card interface by connection and activation of the contacts by the interface device (Vcc then RAZ according to ISO 7816 specification). The command also resets the card. This command is only allowed once the card reader is connected in this session. The response is 'free' if the card reader is not connected, 'busy' if it is connected in another session, 'answ_to_reset' if a card is inserted and performed a reset correctly, and 'no_answ_to_reset' if there is no card, or the card did not reset properly.

- **power_off_card**: powers off the card interface according to the ISO 7816 specification. The response is 'connected' if the session is connected & the operation succeeded, 'free' if the session is not connected, and 'busy' if another session is connected.

- **reset_card**: resets the card interface when the smart card is switched on. The responses are the same as for power_on_card.

- **read_status**: reads the current connection and card status. It is not necessary to connect to issue this command.

- **read_answ_to_reset**: gets the answer-to-reset message for the current card, if one is in the socket. It is not necessary to connect to issue this command. The response is 'answ_to_reset' if one is available, otherwise 'no_answ_to_reset'.

B.3.1.2 Smart Card Reply

Syntax		No. of Bits	Mnemonic
smart_card_reply() {			
smart_card_reply_tag		**24**	**uimsbf**
length_field ()			
smart_card_reply_id		**8**	**uimsbf**
smart_card_status	**8**	**uimsbf**	
if (smart_card_reply_id == answ_to_reset) {			
for (i=0; i < n; i++) {			
char_value	**8**	**uimsbf**	
}			
}			
}			

smart_card_reply_id	id value
connected	01
free	02
busy	03
answ_to_reset	04
no_answ_to_reset	05
reserved	other values

smart_card_status	value
card_inserted	01
card_removed	02
card_in_place_power_off	03
card_in_place_power_on	04
no_card	05
unresponsive_card	06
refused_card	07
reserved	other values

Smart Card Reply is returned in response to a Smart Card Cmd object, or in response to a Smart Card Send object if send is an inappropriate operation at that point. It is also sent unrequested on the connected session (if any) when a card is removed or inserted to signal the fact. The reply id values are:

- **connected**: the response on a connected session where answ_to_reset or no_answ_to_reset does not need to be sent.
- **free**: sent in response to a ‘disconnect’ command, or to other commands if the smart card reader is not connected.
- **busy**: sent in response to commands not allowed to a smart card reader already connected on another session.
- **answ_to_reset**: sent with the answer-to-reset message from the card.
- **no_answ_to_reset**: sent when answer-to-reset has been requested but is not available.

Each reply also carries the card status at that time. The following status values are used:

- **card_inserted**: sent unrequested on a connected session when a card is inserted.
- **card_removed**: sent unrequested on a connected session when a card is removed.
- **card_in_place_power_off**: sent in all replies when a card is inserted and recognised and the card is not powered.
- **card_in_place_power_on**: sent in all replies when a card is inserted and recognised and the card is powered.
- **no_card**: sent in all replies when no card is plugged in.
- **unresponsive_card**: in all replies after the plugged in card does not reply to a reset or if it fails to respond to a Smart Card Send.
- **refused_card**: sent in all replies after a card responds to a reset, but not in an expected way.

B.3.1.3 Smart Card Send

Syntax	No. of Bits	Mnemonic
smart_card_send () {		
smart_card_send_tag	**24**	**uimsbf**
length_field ()		
CLA	**8**	**uimsbf**
INS	**8**	**uimsbf**
P1	**8**	**uimsbf**
P2	**8**	**uimsbf**
length_in	**16**	**uimsbf**
for (i=0; i<length_in; i++)		
char_value	**8**	**uimsbf**
}		
length_out	**16**	**uimsbf**
}		

This is used to send data to a connected card which has completed a reset cycle. The response is either a Smart Card Rcv object with data and/or the status bytes SW1 and SW2, or a Smart Card Reply with status information if it is not appropriate to send data to the card at that point.

B.3.1.4 Smart Card Rcv

Syntax	No. of Bits	Mnemonic
smart_card_rcv () {		
smart_card_rcv_tag	**24**	**uimsbf**
length_field ()		
for (i=0; i < n; i++) {		
char_value	**8**	**uimsbf**
}		
SW1	**8**	**uimsbf**
SW2	**8**	**uimsbf**
}		

This is sent in response to Smart Card Send on a connected session with an operating card.

B.3.2 Smart Card Reader resource coding

Resource	class	type	version	resource identifier
Smart card reader	112	see §B.3.2.1	1	00700041

apdu_tag	tag value (hex)	Resource	Direction host<-->app
$T_{smart\_card\_cmd}$	9F 8E 00	Smart Card reader	<---
$T_{smart\_card\_reply}$	9F 8E 01	Smart Card reader	--->
$T_{smart\_card\_send}$	9F 8E 02	Smart Card reader	<---
$T_{smart\_card\_rcv}$	9F 8E 03	Smart Card reader	--->

B.3.2.1 Resource type coding

The Smart Card Reader resource may have up to 16 instances, supplied either by the host or by other modules. The application selects the required reader by specifying the required device number as the four least significant bits of the resource type field, as shown in Figure B.1.

bit	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	device no.			

Figure B.1: Smart Card reader resource type structure

B.4 DVB EPG Future Event Support Class

This resource is made available by an EPG application to support communication of entitlement information from CA modules to Electronic Programme Guides - whether host-based or module-based. CA modules which offer entitlement information to EPG applications shall create a session to all instances of this resource indicated by the Resource Manager and shall operate the protocol as required by the EPG application.

The protocol uses two objects - an event_enquiry object and an event_reply object. The protocol allows an EPG to enquire of the entitlement status of a current or future event, identified by its Event ID in the Service Information. The CA module can reply indicating that entitlement for the event is available, is not available, may be available after dialogue (e.g. pay-per-view), or its entitlement status is unknown. Additionally the protocol supports a process for allowing a CA module to start a dialogue to make a potential entitlement available, e.g. for pay-per-view, and then returning to the EPG dialogue. This resource does not prevent EPG and CA applications from the same or co-operating providers using the private resource mechanism to integrate operation of EPG and CA more completely. However CA modules wishing to provide entitlement information to independent EPGs shall use this resource.

B.4.1 Objects

B.4.1.1 Event Enquiry Object

This is a query from EPG to CA module.

Table B.1: Event Enquiry object coding

Syntax	No. of bits	Mnemonic
event_enquiry () {		
event_enquiry_tag	**24**	**uimsbf**
length_field()		
event_cmd_id	**8**	**uimsbf**
network_id	**16**	**uimsbf**
original_network_id	**16**	**uimsbf**
transport_stream_id	**16**	**uimsbf**
service_id	**16**	**uimsbf**
event_id	**16**	**uimsbf**
}		

event_cmd_id

This can have the following values:

event_cmd_id	value
mmi	02
query	03
reserved	other values

The 'mmi' command allows the CA module to start a MMI dialogue to make entitlement available. It is typically used after the EPG has already determined (by a previous 'query' command) that an MMI session is required by the CA module, and has closed its own MMI session temporarily to make the MMI resource available to the CA module. The 'query' command is used to just ask for the current entitlement status of an event without requesting a dialogue.

The event ID is fully qualified by all the other location identifiers in SI which make it unique.

B.4.1.2 Event Reply Object

This is a reply from CA module to EPG.

Table B.2: Event Reply object coding

Syntax	No. of bits	Mnemonic
event_reply () {		
event_reply_tag	**24**	**uimsbf**
length_field()		
event_status	**8**	**uimsbf**
}		

event_status

This can have the following values:

event_status	value
entitlement_unknown	00
entitlement_available	01
entitlement_not_available	02
mmi_dialogue_required	03
mmi_complete_unknown	04
mmi_complete_available	05
mmi_complete_not_available	06
reserved	other values

'entitlement_unknown' means that the CA module cannot determine the entitlement status of this event. 'entitlement_available' means that entitlement for this event is currently available. 'entitlement_not_available' means that entitlement for this event is not currently available and cannot be made available by any user dialogue with the CA module. 'mmi_dialogue_required' indicates that entitlement is not currently available but could be made available after a dialogue between CA and user. 'mmi_complete_unknown' is returned in response to a event_enquiry with the 'mmi' command after the CA mmi dialogue is complete and the entitlement status is still unknown. 'mmi_complete_available' is returned in response to a event_enquiry with the 'mmi' command after the CA mmi dialogue is complete and the entitlement has been granted. 'mmi_complete_not_available' is returned in response to a event_enquiry with the 'mmi' command after the CA mmi dialogue is complete and the entitlement has not been granted, for example if the user decided, in the dialogue, not to purchase the event after all.

Note that the event_reply object returned refers to the preceding event_enquiry object received, so the EPG must not send another event_enquiry object until it receives the event_reply for the current enquiry.

B.4.1.3 Example of Use

A typical protocol exchange is described. This uses an enhancement of the MMI resources allowing the application to close a MMI session with a short delay before returning to the current service to allow another application to start its own dialogue seamlessly with the previous one.

EPG: event_query(query, event x)
CA: event_reply(mmi_dialogue_required)
EPG: close_mmi(delay); remember context
EPG: event_query(mmi, event x)
CA: open mmi; perform dialogue
CA: close_mmi(delay)
CA: event_reply(mmi_complete_available) (may also reply not_available or unknown)
EPG: open mmi; continue EPG dialogue from remembered context

B.4.2 EPG Future Event Support resource coding

Resource	class	type	version	resource identifier
EPG Future Event Support	120	see §B.4.2.1	1	00780041

apdu_tag	tag value (hex)	Resource	Direction EPG<-->CA
$T_{event\_enquiry}$	9F 8F 00	EPG Future Evt Supp	--->
T_{event_reply}	9F 8F 01	EPG Future Evt Supp	<---

B.4.2.1 Resource type coding

It may be that there is more than one EPG on a host. For example the host may support an internal EPG application and the user may have also purchased a module-based EPG. In this case the host shall allocate resource type numbers, beginning at zero, to distinguish the instances of EPG running. CA modules which support EPGs using this resource shall create a session to each instance of this class advertised as a different resource type by the Resource Manager.