TECHNICAL SPECIFICATION

ISO/TS 18234-2

Second edition
2013-10-15

Corrected version
2013-11-01

Intelligent transport systems — Traffic and travel information via transport protocol experts group, generation 1 (TPEG1) binary data format —

Part 2: Syntax, semantics and framing structure (TPEG1-SSF)

Systèmes intelligents de transport — Informations sur le trafic et le tourisme via les données de format binaire du groupe d'experts du protocole de transport, génération 1 (TPEG1)

Partie 2: Structure de syntaxe, de sémantique et de cadrage (TPEG1-SSF)

Reference number
ISO/TS 18234-2:2013(E)

COPYRIGHT PROTECTED DOCUMENT

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

Published in Switzerland

Contents

Page

Foreword

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

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

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

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

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

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

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

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

ISO/TS 18234-2 was prepared by the European Committee for Standardization (CEN) Technical Committee CEN/TC 278, *Road transport and traffic telematics,* in collaboration with ISO Technical Committee ISO/TC 204, *Intelligent transport systems*, in accordance with the Agreement on technical cooperation between ISO and CEN (Vienna Agreement).

This second edition cancels and replaces the first edition (ISO/TS 18234-2:2006). Clauses 5, 6 and 7 have been technically revised.

ISO/TS 18234 consists of the following parts, under the general title *Intelligent transport systems — Traffic and travel information via transport protocol experts group, generation 1 (TPEG1) binary data format*:

— *Part 1: Introduction, numbering and versions (TPEG1-INV)*

— *Part 2: Syntax, semantics and framing structure (TPEG1-SSF)*

— *Part 3: Service and network information(TPEG1-SNI)*

— *Part 4: Road Traffic Message application (TPEG1-RTM)*

— *Part 5: Public Transport Information (PTI) application*

— *Part 6: Location referencing applications*

— *Part 7: Parking information (TPEG1-PK1)*

— *Part 8: Congestion and travel-time application (TPEG1-CTT)*

— *Part 9: Traffic event compact (TPEG1-TEC)*

— *Part 10: Conditional access information (TPEG1-CAI)*

— *Part 11: Location Referencing Container (TPEG1-LRC)*

This corrected version of ISO 18234-2:2013 incorporates the following corrections:

— The quality of Figures 4 and 5 has been improved for legibility.

Introduction

TPEG technology uses a byte-oriented data stream format, which may be carried on almost any digital bearer with an appropriate adaptation layer. TPEG messages are delivered from service providers to end-users, and are used to transfer application data from the database of a service provider to a user's equipment.

This Technical Specification describes the Service and Network Information Application, which provides a means of informing end-users about all possible services and their content which are considered relevant by a service provider to either provide continuity of his services or inform the end-user about other related services. As stated in the design criteria, TPEG is a bearer independent system. Therefore some rules are established for the relation of information contents of the same service on different bearers. Also the mechanisms for following a certain service on one single bearer have to be defined. For the receiver it is essential to find an adjacent or similar service if it leaves the current reception area. Nonetheless, basic information describing the service itself is necessary. For the ease of the user, e.g. the service name, the service provider name, the operating time and many other hints are delivered by the TPEG-SNI application.

General models for the hand-over and the referencing of services are developed and shown in detail. It is important to note that this Technical Specification is closely related to ISO/TS 18234-3 and thus they are dependent upon each other and must be used together.

The brief history of TPEG technology development dates back to the European Broadcasting Union (EBU) Broadcast Management Committee establishing the B/TPEG project group in autumn 1997 with the mandate to develop, as soon as possible, a new protocol for broadcasting traffic and travel-related information in the multimedia environment. TPEG technology, its applications and service features are designed to enable travel-related messages to be coded, decoded, filtered and understood by humans (visually and/or audibly in the user's language) and by agent systems.

One year later in December 1998, the B/TPEG group produced its first EBU specifications. Two Technical Specifications were released. ISO/TS 18234-2, this document, described the Syntax, Semantics and Framing Structure, which is used for all TPEG applications. ISO/TS 18234-4 (TPEG-RTM) described the first application, for Road Traffic Messages.

Subsequently, CEN/TC 278/WG 4, in conjunction with ISO/TC 204, established a project group comprising the members of B/TPEG and they have continued the work concurrently since March 1999. Since then two further parts were developed to make the initial complete set of four parts, enabling the implementation of a consistent service. ISO/TS 18234-3 (TPEG-SNI) describes the Service and Network Information Application, which should be used by all service implementations to ensure appropriate referencing from one service source to another. ISO/TS 18234-1 (TPEG-INV), completes the series, by describing the other parts and their relationship; it also contains the application IDs used within the other parts.

In April 2000, the B/TPEG group released revised Parts 1 to 4, all four parts having been reviewed and updated in the light of initial implementation results. Thus a consistent suite of specifications, ready for wide scale implementation, was submitted to the CEN/ISO commenting process.

In November 2001, after extensive response to the comments received and from many internally suggested improvements, all four parts were completed for the next stage: the Parallel Formal Vote in CEN and ISO. But a major step forward has been to develop the so-called TPEG-Loc location referencing method, which enables both map-based TPEG-decoders and non map-based ones to deliver either map-based location referencing or human readable information. ISO/TS 18234-6 is now a separate specification and is used in association with the other parts of ISO/TS 18234 to provide comprehensive location referencing. Additionally, ISO/TS 18234-5, has been developed and been through the commenting process.

This Technical Specification provides a full specification to the primitives used, framing, time calculation, numbers and to specific rules such as CRC calculation.

During the development of the TPEG technology a number of versions have been documented and various trials implemented using various versions of the specifications. At the time of the publication of this Technical Specification, all parts are fully inter-workable and no specific dependencies exist.

This Technical Specification has the technical version number TPEG-SSF_3.0/003.

Intelligent transport systems — Traffic and travel information via transport protocol experts group, generation 1 (TPEG1) binary data format —

Part 2:
Syntax, semantics and framing structure (TPEG1-SSF)

1 Scope

This Technical Specification establishes the method of referencing used within a TPEG data-stream to allow a service provider to signal availability of the same service on another bearer channel or similar service data from another service.

TPEG is a byte-oriented stream format, which may be carried on almost any digital bearer with an appropriate adaptation layer. TPEG messages are delivered from service providers to end-users, and are used to transfer application data from the database of a service provider to a user's equipment.

The protocol is structured in a layered manner and employs a general purpose framing system which is adaptable and extensible, and which carries frames of variable length. This has been designed with the capability of explicit frame length identification at nearly all levels, giving greater flexibility and integrity, and permitting the modification of the protocol and the addition of new features without disturbing the operation of earlier client decoder models.

2 Normative references

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

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

ISO/IEC 8859-1, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

ISO/IEC 8859-2, *Information technology — 8-bit single-byte coded graphic character sets — Part 2: Latin alphabet No. 2*

ISO/IEC 8859-3, *Information technology — 8-bit single-byte coded graphic character sets — Part 3: Latin alphabet No. 3*

ISO/IEC 8859-4, *Information technology — 8-bit single-byte coded graphic character sets — Part 4: Latin alphabet No. 4*

ISO/IEC 8859-5, *Information technology — 8-bit single-byte coded graphic character sets — Part 5: Latin/Cyrillic alphabet*

ISO/IEC 8859-6, *Information technology — 8-bit single-byte coded graphic character sets — Part 6: Latin/Arabic alphabet*

ISO/IEC 8859-7, *Information technology — 8-bit single-byte coded graphic character sets — Part 7: Latin/Greek alphabet*

ISO/IEC 8859-8, *Information technology —8-bit single-byte coded graphic character sets — Part 8: Latin/Hebrew alphabet*

ISO/IEC 8859-9, *Information technology — 8-bit single-byte coded graphic character sets — Part 9: Latin alphabet No. 5*

ISO/IEC 8859-10, *Information technology — 8-bit single-byte coded graphic character sets — Part 10: Latin alphabet No. 6*

ISO/IEC 8859-13, *Information technology — 8-bit single-byte coded graphic character sets — Part 13: Latin alphabet No. 7*

ISO/IEC 8859-14, *Information technology — 8-bit single-byte coded graphic character sets — Part 14: Latin alphabet No. 8 (Celtic)*

ISO/IEC 8859-15, *Information technology — 8-bit single-byte coded graphic character sets — Part 15: Latin alphabet No. 9*

ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*

3 Abbreviated terms

For the purposes of this document, the following abbreviated terms apply:

AID	Application Identification
BPN	Broadcast, Production and Networks (an EBU document publishing number system)
B/TPEG	Broadcast/TPEG (the EBU project group name for the specification drafting group)
CEN	Comité Européen de Normalisation
DAB	Digital Audio Broadcasting
DARC	Data Radio Channel - an FM sub-carrier system for data transmission
DVB	Digital Video Broadcasting
EBU	European Broadcasting Union
INV	Introduction, Numbering and Versions (see ISO/TS 18234-1)
IPR	Intellectual Property Right(s)
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union - Telecom
OSI	Open Systems Interconnection
RTM	Road Traffic Message application (see ISO/TS 18234-4)

SNI	Service and Network Information application (see ISO/TS 18234-3)
SSF	Syntax, Semantics and Framing Structure (this Technical Specification)
TPEG	Transport Protocol Expert Group
TTI	Traffic and Travel Information
UAV	unassigned value
UTC	Coordinated Universal Time

4 Design principles

The following principles have been assumed in the development of the TPEG protocol, structure and semantics:

— TPEG is unidirectional;

— TPEG is byte-oriented, where a byte is represented by eight bits;

— TPEG provides a protocol structure, which employs asynchronous framing;

— TPEG includes a CRC error detection capability applicable on a variety of different levels;

— TPEG assumes the use of a transparent data channel;

— TPEG assumes that underlying systems will have an appropriate level of reliability;

— TPEG assumes that underlying systems may employ error correction;

— TPEG has a hierarchical data frame structure;

— TPEG is used to transport information from database to database;

— TPEG provides service provider name, service name and network information;

— TPEG permits the use of encryption mechanisms, if required by an application.

4.1 TPEG transmission

TPEG is intended to operate via almost any simple digital data channel, and it assumes nothing of the channel other than the ability to convey a stream of bytes. To this end, the concept of transmission via a “piece of wire” is envisaged, in which the bearer has no additional service features.

In Figure 1, a variety of possible transmission channels are shown. The only requirement of the channel is that a sequence of bytes may be carried between the TPEG generator and the TPEG decoder. This requirement is described as “transparency”. However it is recognized that data channels may introduce errors. Bytes may be omitted from a sequence, bytes may become corrupted or additional and erroneous data could be received. Therefore TPEG incorporates error detection features at appropriate points and levels. It is assumed that bearer systems will introduce an appropriate level of error correction.

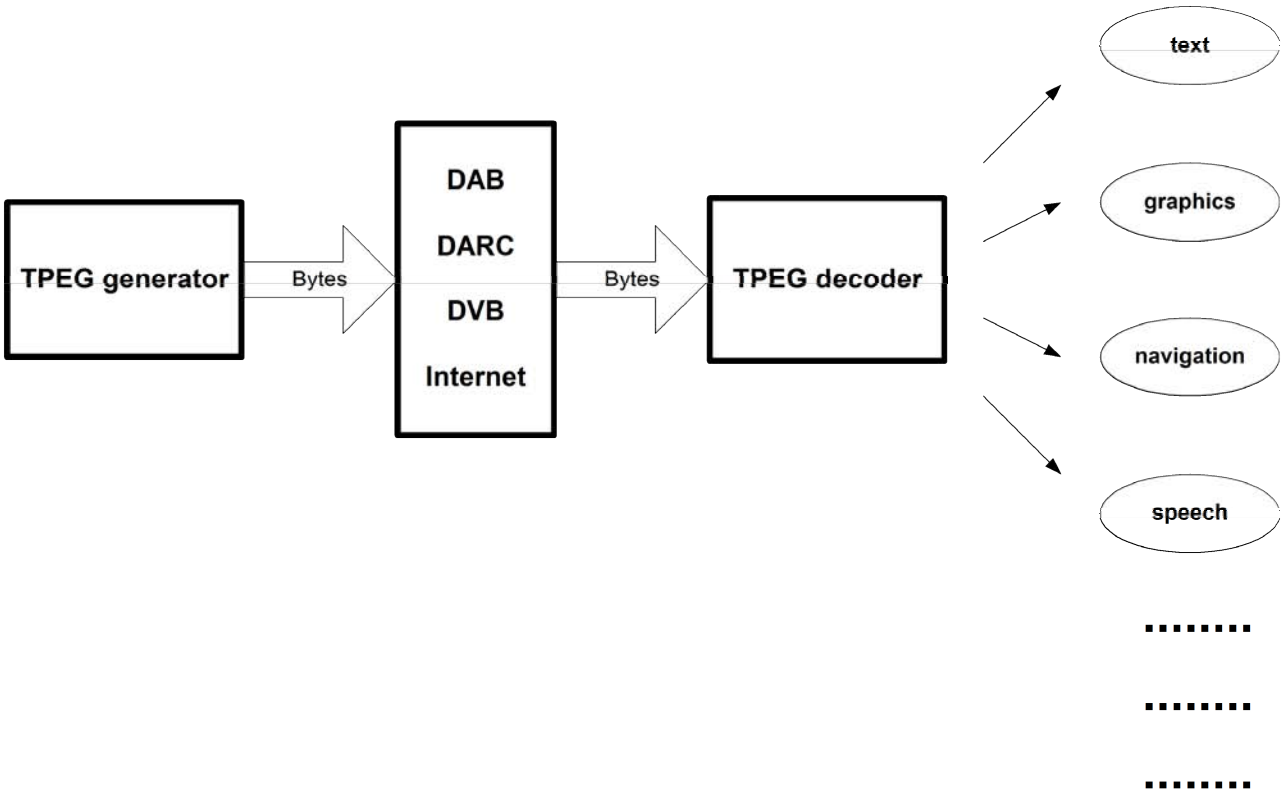


Figure 1 — TPEG data may be delivered simultaneously via different bearer channels

4.2 TPEG layer model

In Figure 2, the different layers of the TPEG protocol are identified in accordance with the ISO/OSI model (ISO/IEC 7498-1).

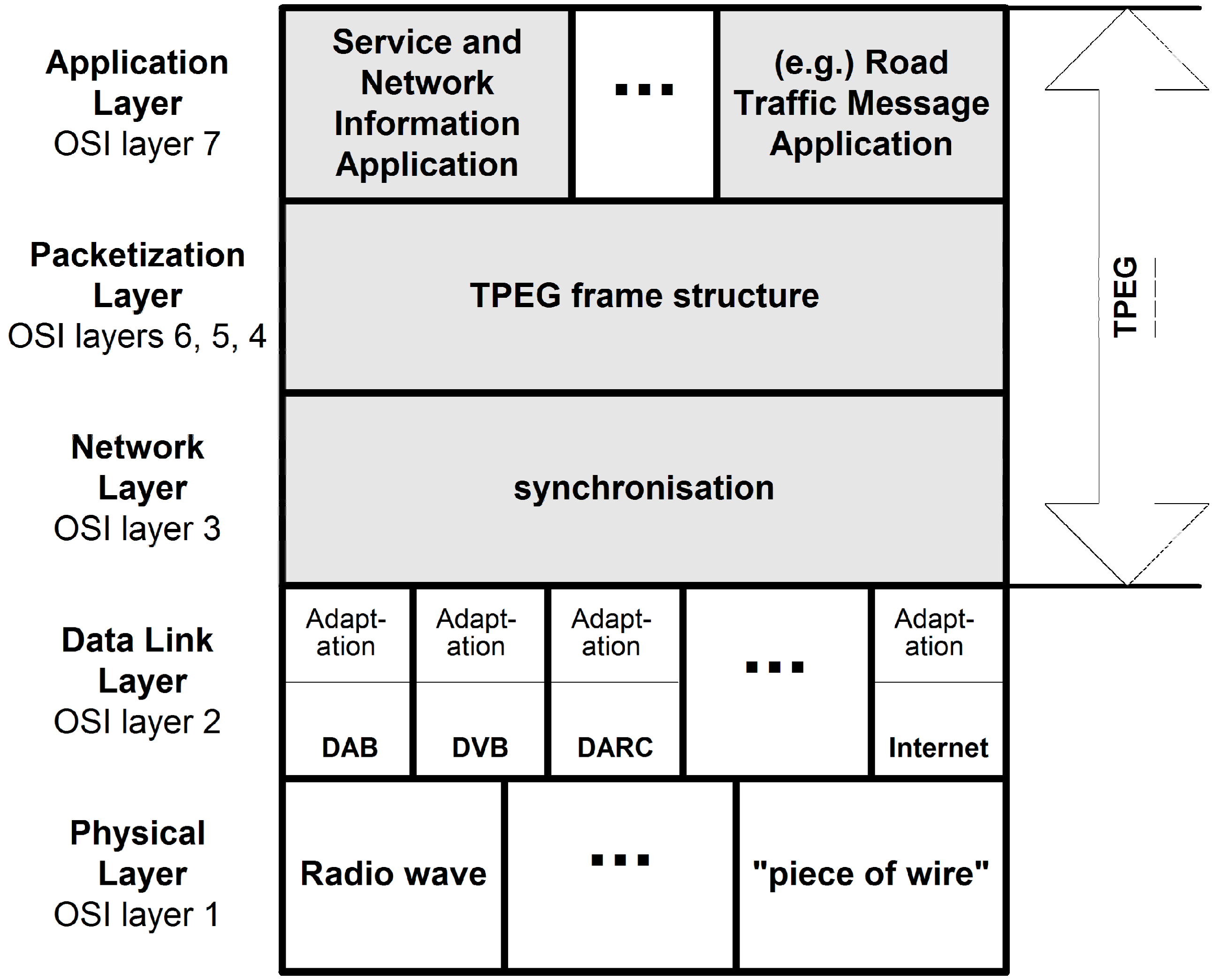


Figure 2 — TPEG in relation to the ISO/OSI Layer Model via different bearer channels

Layer 7 is the top level and referred to in TPEG as the application layer. Initially the following applications were defined:

— TPEG specifications - Part 3: Service and Network Information Application (Service provider name, logo, hand-over information, etc.) (CEN ISO/TS 18234-3);

— TPEG specifications - Part 4: Road Traffic Message application (Event description, location description, etc.) (CEN ISO/TS 18234-4).

Layer 4 is the packetization layer. Components are merged into a single stream and encrypted and/or compressed.

Layer 3 is the network layer. This layer defines the means for synchronization and routing. This is the lowest layer of the TPEG protocol.

Layer 2 is the datalink layer. This layer consists of a wide range of different bearers, which are suitable carriers for the TPEG protocol. An adaptation layer may be required in order to map the TPEG stream onto that bearer.

Layer 1 is the physical layer. This defines the transmission medium (radio waves, wire, optical, etc.). One particular bearer can make use of different physical layers.

5 Conventions and symbols

5.1 Conventions

5.1.1 Byte ordering

All numeric values using more than one byte are coded in “Big Endian” format (most significant byte first). Where a byte is subdivided into bits, the most significant bit (“b7”) is at the left-hand end and the least significant bit (“b0”) is at the right-hand end of the structure.

5.1.2 Method of describing the byte-oriented protocol

TPEG uses a data-type representation for the many structures that are integrated to form the transmission protocol. This textual representation is designed to be unambiguous, easy to understand and to modify, and does not require a detailed knowledge of programming languages.

Data types are built up progressively. Primitive elements, which may be expressed as a series of bytes are built into compound elements. More and more complex structures are built up with compound elements and primitives. Some primitives, compounds and structures are specified in this Technical Specification, and apply to all TPEG Applications. Other primitives, compounds and structures are defined within applications and are local only to that application.

A resultant byte-stream coded using C-type notation is shown in CEN ISO/TS 18234-2:2006, Annex E.

5.1.3 Reserved data fields

If any part of a TPEG data structure is not completely defined, then it should be assumed to be available for future use. The notation is UAV (unassigned value). This unassigned value should be encoded by the service provider as the value 00 hex. This allows newer decoders using a future TPEG Standard to ignore this data when receiving a service from a provider encoding to this older level of specification. A decoder which is not aware of the use of any former UAVs can still make use of the remaining data fields of the corresponding information entity. However, the decoder will not be able to process the newly defined additional information.

5.2 Symbols

5.2.1 Literal numbers

Whenever literal numbers are quoted in TPEG Standards, the following applies:

123 = 123 decimal

123 hex = 123 hexadecimal

5.2.2 Variable numbers

Symbols are used to represent numbers whose values are not predefined within the TPEG Standards. In these cases, the symbol used is always local to the data type definition. For example, within the definition of a data type, symbols such as “n” or “m” are often used to represent the number of bytes of data within the structure, and the symbol “id” is used to designate the occurrence of the identifier of the data type.

5.2.3 Implicit numbers

Within the definition of a data structure it is frequently necessary to describe the inclusion of a component which is repeated any number of times, zero or more. In many of these cases it is convenient to use a numerical symbol to show the component structure being repeated a number of times, but the number itself is not explicitly included within the definition of the data structure. Often, the symbol “m” is used for this purpose.

6 Representation of syntax

6.1 General

This clause introduces the terminology and the syntax that is used to define TPEG data elements and structures.

6.2 Data type notation

6.2.1 Rules for data type definition representation

The following general rules are used for defining data types:

— a data type is written in upper camel case letters in one single expression.[1] The data type may contain letters (a-z), number (0-9), underscore "_", round brackets "()" and colon ":"; the first must be a letter;

EXAMPLE 1 IntUnLo stands for Integer Unsigned Long

— a data type is framed by angle brackets “ **< >** ” ;

— the content of a data type is defined by a colon followed by an equal sign “ **:=** ”;

— the end of a data type is indicated by a semicolon “ **;** ”;

— a descriptor written in lower camel case may be added to a data type as one single expression without spaces;

— a descriptor is framed by round brackets “ ( ) ”;

— the descriptor contains either a value or a name of the associated type;

— data types in a definition list of another one are separated by commas “ **,** ”. The order of definition is defined as the order of occurrence in a data stream;

— curly brackets (braces) “ { } ” group together a block of data types;

— control statements (“if”, “infinite”, “unordered” or “external”) are noted in lower case letters. A control statement is followed by a block statement or only one data type:

  1) “if” defines a condition statement. The block’s (or data type’s) occurrence is conditional to the condition statement being valid. The condition statement is framed with round brackets. This statement applies to any data type;

  2) “infinite” defines endless repetition of the block (or data type). This is only used to mark the main TPEG stream as not ending stream of data;

[1] Camel case is the description given to the use of compound words wherein each individual word is signalled by a capital letter inside the compound word. Upper camel case means that the compound word begins with an upper-case (capital) letter, and lower camel case means the compound word begins with a small letter.

3) “unordered” defines that the following block contains data types which may occur in any order, not only the one used to specify subsequent data types. This statement applies to components only. (See Clause A2.3.3 - Components);

4) “external” defines that the content of the data type is being defined external to the scope of given specification. The control statement “external” must be followed by only one data. A reference to the corresponding specification should follow in the comment. All types specified in TYP specification are treated as being in scope of any application

EXAMPLE 2

<MMCLink(1)>:=	: externally defined component
external <MessageManagementContainer(1)>;	: id = 1, See Annex B (Message Management Container)

— the expression “ n * ” indicates multiplicity of occurrence of a data type . The lower and upper bound are implicitly from 0 to infinite; other bounds are described in square brackets between two points " .. " and behind the data type descriptor. The " * " stands for no limitation at upper bound

EXAMPLE 3

m * **<IntUnTi>**(Attribute) [1..*] ,	: The “Attribute” must occur once at least and up to infinite.

— a function “ f_n ( ) ” that is calculated over a data type is indicated by italic lower case letters. The comment behind the definition of the function shall explain which function is used;

— any text after a colon “ : ” is regarded as a comment;

— a data type definition can be a *template* (i.e. not fully defined declarative structure) having a *parameter* inside of round brackets "(x)" at the end of the data type name. Templates define structures, whose structural definition is included as a basis for other data type definitions. To declare the given template (making it identifiable) the name of the parameter is repeated as a descriptor in a nested data type of the subsequent definition list. Templates allow for reading the generalised part of different instances i.e. to specify data type interfaces. (See Clause A2.3.2 - Using templates as interfaces for further description)

EXAMPLE 4

<Template(x)> :=	: x defines the template parameter
<IntUnTi>(x);	: descriptor x defines position of setting the parameter in the list

— a data type can *inherit* a template by concatenating the data type name of the template including the square brackets to its own name. The data type itself can again be a template having the "(x)" at its end of name, or it instantiates the inherited template by defining the value of the parameter in the brackets. In the latter case the brackets shall contain the **decimal** number of the identifier and the value shall be set in the subsequent definition list. The structural definition of the inherited template is repeated as the first part of the definition list before new data types are specified. (See Clause A2.3.2 - Using templates as interfaces for further description)

EXAMPLE 5

<AnotherTemplate(x)<Template(x)>>:=	: second template inherits first
<IntUnTi>(x),	: repeated definition from 1st template
<IntUnLi>(n);	: additional structural definition
<Instance<AnotherTemplate(1)>>:=	**: instantiation of the second template**
<IntUnTi>(1),	: definition of parameter in the stream
<IntUnLi>(n),	: structural definition from template
<IntUnTi>(value);	: some more definition

— in the definition list a specific instance of a template (i.e. declarative structure) is described without the brackets. Any inherited data type of this template may occur at that position in the data stream

EXAMPLE 6

<SomeData>:=	
<AnotherTemplate>(anyAnotherTemplate);	: Data stream contains e.g. <**Instance**>

The following additional guidelines help to improve the readability of data type definitions:

— data type names are written in bold;

— nested data type definitions are defined from top to bottom (i.e. higher levels first, then lower levels);

— a box is drawn around a data type definition;

— for clear graphical presentation, lines in a coding box if they are too long to fit, are broken with a backslash "\" followed by a carriage return. The broken line restarts with an additional backslash

EXAMPLE 7

<LongLinesExample>:=	
<DateTimeVeryLongType	: First line
\NameMayBeInSeveralLines>,	: Second line
<DateTime>,	
<ShortString>;	

6.2.2 Description of data type definition syntax

A data type is an interpretation of one or more bytes. Each data type has a structure, which may describe the data type as a composition of other defined data types. The data type structure shows the composition and the position of each data element. TPEG defines data structures in the following manner:

<NewDataType>:=	: Description of data type
<DataTypeA>(descriptorA)**,**	: Description of data A
<DataTypeB>(descriptorB);	: Description of data B

This shows an example data structure, which has just two parts, one of type **<DataTypeA>** and the other of **<DataTypeB>**. A descriptor may be assigned to the data type, to relate the element to another part of the definition. Comments about the data structure are included at the right-hand side delimited by the colon ":" separator. Each of the constituent data types may be itself composed of other data types, which are defined separately. Eventually each data type is expressible as one or more bytes.

Where a data structure is repeated a number of times, this may be shown as follows:

<NewDataType>:=	: Description of data type
<DataTypeA>,	: Description of data A
m * **<DataTypeB>**[0..*];	: Description of data B

Often, in such cases it is necessary to explicitly deliver to the decoder the number of times a data type is repeated; sometimes it is not, because other means like framing or internal length coding allows knowledge of the end of the list of the repeated data type. In other cases the overall length of a data structure in bytes needs to be specified. Additionally the constraint on occurrences can be added, which tells how many instances of the data type must be expected by the decoder. The "*" as upper bound means in this case that at this place no restriction is given to the upper bound; in other words, infinite elements may follow.

Where the number of repetitions must be signalled, it may be accomplished using another data element as follows:

<NewDataType>:=	: Description of data type
<IntUnTi>(n)**,**	: An integer representing the value of "n"
n * **<DataTypeA>**[0..255]**,**	: Description of data A
<DataTypeB>;	: Description of data B

In the above example a decoder has to have the value of "n" in order to correctly determine the n'th position of the **<DataTypeB>** in the list. Here as consequence of data type IntUnTi not more as 255 instances of the data type can be coded.

In the following example the decoder uses the value of "n" to determine the overall length of the data structure, and the value of "m" determines that **<DataTypeB>** is repeated m times:

<NewDataType>:=	: Description of data type
<IntUnTi>(n)**,**	: Length, n, of data structure in bytes
m * **<DataTypeA>;**	: Description of data A

This data type definition is used to describe a variable structure switched by the value of x:

<NewDataType>:=	: Description of data type
<IntUnTi>(x)**,**	: Select parameter, x
if (x=1) then **<DataTypeA>,**	: Included if x equals 1
if (x=2) then **<DataTypeB>,**	: Included if x equals 2

6.3 Application dependent data types

This clause describes the methodology and syntax by which application data types may be constructed within TPEG Applications. Two basic forms are described: data structures (being non-declarative) and components (being declarative). Components contain an identifier which labels the structure, and which can be used by a decoder to determine the definition of content of the structure. As such, components are used where options are required, or where an application needs to build in 'future proofing'. Data structures do not contain such information, and are used in all other positions.

This Annex does not specify the structures, which are actually used in TPEG Applications. Such specifications are made in the respective parts of the Standard. However examples are given to show how such structures may be built from the primitive elements already described above.

6.3.1 Data structures

Data structures are built up from several (i.e. more than one) elements: primitive, compound or other structures (both non-declarative and declarative). As such, any application specific data type definition having no component identifier is per definition a data structure. The term data structure is specifically used for data type definitions having more than one sub element defined.

Examples of data structure might be:

EXAMPLE 1

<Activity>:=	: Activity
<DateTime>,	: Beginning
<DateTime>,	: End
<ShortString>;	: Text

EXAMPLE 2

<Wave>:=	: Sound sample
<IntUnLi>(n)**,**	: Length of samples, n
n * **<IntSiTi>**(sample)[0..8000];	: Between 0 and 8000 occurrences of a sample

Another example making use of a condition within a data type definition is shown below.

EXAMPLE 3 An application could use the example data types above in the following way

<Appointment>:=	: Appointment
<IntUnTi>(at)**,**	: Alarm type
if (at = 1)	
<WaveAlarm>,	: Remind with a sound
if (at = 2)	
<TextAlarm>,	: Remind with a text
<Activity>;	: Let some action follow

<WaveAlarm>:=	: Sound alarm
<DateTime>,	: When to wake up
<Wave>;	: Sound to wake up to!

<TextAlarm>:=	: Text alarm
<DateTime>,	: When to display
<ShortString>;	: Text to display

For optional values a general mechanism is provided, using a bitarray for signalling optional values. In the case that a corresponding bit of the bitarray is set (=1), the optional attribute is stored in the stream. In case the bit is unset the attribute is not available and the next following attribute shall be processed in the stream.

EXAMPLE 4 Data structure with optional elements, signalled by a preceding bitarray as selector

<TimeInterval>:=	
<BitArray>(selector),	: DaySelector
if (bit 0 of selector is set)	
<IntUnTi>(years),	: Number of years between 0 and 100
if (bit 1 of selector is set)	
<IntUnTi>(months),	: Number of months between 0 and twelve
if (bit 2 of selector is set)	
<IntUnTi>(days),	: Number of days between 0 and 31
if (bit 3 of selector is set)	
<IntUnTi>(hours),	: Number of hours between 0 and 24
if (bit 4 of selector is set)	
<IntUnTi>(minutes),	: Number of minutes between 0 and 60
if (bit 5 of selector is set)	
<IntUnTi>(seconds);	: Number of seconds between 0 and 60

6.3.2 Using templates as interfaces

In addition to the possibility of coding the complete and static structural definition of a data structure, the syntax does foresee that parts of the structure are conditionally different; signalled by a well defined first part some other data types are different.

EXAMPLE

A tagged value (also known as TagLengthValue-Coding) starts with a type and length; afterwards the value follows. Let's assume the type is an enumeration of some possible values, one would first specify the interface having only the type defined. The different tagged value types would now inherit this interface, i.e. would have the type defined as first element amended with the definition of the tagged value data type. The decoder now reads the interface information (the type attribute) and knows how to proceed for reading the rest of the tagged value from the stream.

<DifferentDataList>:=	: A list of data
n * **<TaggedValue>**(value)**;**	: Different instances can have different types

<TaggedValue(x)>:=	: Template for tagged value
<tav001:ValueType>(type)**,**	: Type of this tagged value
<IntUnTi>(length)**;**	: Length in bytes in case that value type is unknown

Example table **tav001:ValueType**:

Code	**Reference-English 'word'**	**Comment**
001	Service name	
002	Price per month	

Then the resulting list of inherited tagged value data types would be:

<ServiceName<TaggedValue(1)>>:=	: Template for tagged value
<tav001:ValueType>(1)**,**	: Type of this tagged value
<IntUnTi>(length),	: Length in bytes in case that value type is unknown
<**ShortString**>(serviceName)**;**	: Service name

<ServiceName<TaggedValue(2)>>:=	: Template for tagged value
<tav001:ValueType>(2)**,**	: Type of this tagged value
<IntUnTi>(length),	: Length in bytes in case that value type is unknown
<**Float**>(pricePerMonth)**;**	: Price per month

This interface allows a subsequent list of data types which can easily be extended, by using the same interface.

6.3.3 Components

A component is understood as a declarative structure having an interface as described in the previous clause. A decoder of the data stream can identify the content of the structure with the help of the identifier which is unique in the scope of any one TPEG Application Standard. In addition to the identifier a length indicator allows the decoder to step over those components whose ids are unknown to it. This enables the possibility of introducing new components in the data stream although decoders in the market do not know their content. The old decoder does expect the content of the first version of a protocol and ignores simply unrecognized data with small performance loss. The new decoder expects the second version of the protocol and can fully decode that version of the protocol. Components should be used wherever *future extensions* are envisioned, and where 'future proofing' is a strong requirement.

NOTE With this method even non-backwards compatible changes can be introduced into the existing market by having a migration period being backward compatible and then later cutting of not longer supported devices, even though it is expected that the migration will take its time.

In Addition to the concept of declarative structuring a second step of improvement of size efficiency combined with the backward compatibility is specified. The first part following the header of a component in the data stream is defined as *attribute block*. The attribute block starts with the length of the block in bytes which again allows the decoder to step over attributes that are not specified in a first version of the protocol.

The decoder reads the attribute block length and decreases the count of bytes while reading the attributes in case that the last known attribute is read, and the attribute block count is not zero, the remaining bytes in the data stream are omitted to step over to the next well-known part of the data stream.

6.3.3.1 Definition of standard component interface

A component, including attributes, which is the general standard component, containing a unique "generic component id", a length indicating count of bytes following as data after the component length and an attribute length indicating the count of bytes in the attribute block (as first part of the component data). The structure is defined by:

<**Component(x)**>:=	: Component template used for standard components
<IntUnTi>(x),	: id is unique within the scope of the application.
<IntUnLoMB>(compLengthInByte),	: length of the component counted in bytes.
<IntUnLoMB>(attributeBlockLengthInByte);	: length of the attribute block in bytes.

6.3.3.2 Example for jumping over unknown content types

— let C1 be a component with an attribute a1 as ShortInt and a sub component C2;

— let C2 be a component with an attribute a2 as one IntUnTi and a second a3 as ShortString;

— let C3 be a component being the successor of C1.

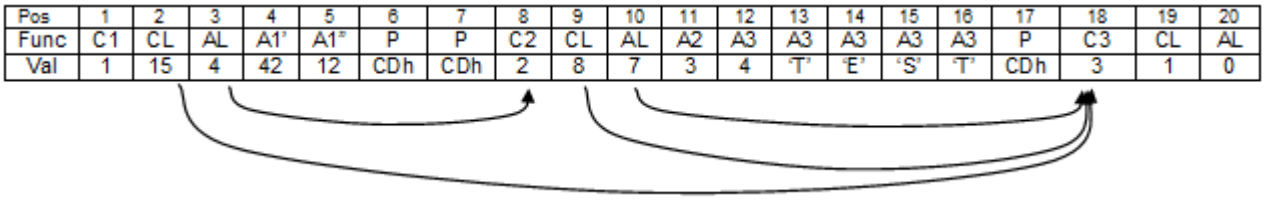
<C1<Component(1)>>:=	
<IntUnTi>(1),	: id = 1.
<IntUnLoMB>(compLengthInByte),	: length of the component counted in bytes
<IntUnLoMB>(attributeBlockLengthInByte);	: length of the attribute block in bytes
<ShortInt>(a1),	: first attribute in C1
<C2>(c2);	: sub component from C1

<C2<Component(2)>>:=	
<IntUnTi>(2),	: id = 2.
<IntUnLoMB>(compLengthInByte),	: length of the component counted in bytes
<IntUnLoMB>(attributeBlockLengthInByte);	: length of the attribute block in bytes
<IntUnTi>(a2),	: first attribute in C2
<ShortString>(a3);	: second attribute in C2

<C3<Component(3)>>:=	
<IntUnTi>(3),	: id = 3.
<IntUnLoMB>(compLengthInByte),	: length of the component counted in bytes
<IntUnLoMB>(attributeBlockLengthInByte);	: length of the attribute block in bytes

For example to demonstrate the method some padding bytes with value CD hex could be added to the stream whereby a decoder could still read C1 – C3. In Figure A.1 one can see a first line with a position number, a second line with the abbreviated function of that byte and a third line with sample content. The arrows under the table show the possible jumps allowing the seeking over the different padding bytes.

Line function abbreviations mean:
CL : component (data) length in bytes — AL : attribute block length in bytes
P : padding bytes — A1, A2, A3 : attributes
C1, C2, C3 : component identifier, begin of the component



Pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Func	C1	CL	AL	A1'	A1"	P	P	C2	CL	AL	A2	A3	A3	A3	A3	A3	P	C3	CL	AL
Val	1	15	4	42	12	CDh	CDh	2	8	7	3	4	'T'	'E'	'S'	'T'	CDh	3	1	0

Figure 3 — Example for jumping over unknown content with component header information

6.4 Toolkits and external definition

Some functionality is shared between different TPEG Applications. This is for example the case for location referencing container and message management container. A TPEG Application therefore can refer to a data type definition not specified in the same Technical Specification.

Toolkits are designed, so that the root components usable as external reference are defined as templates. A TPEG Application using a toolkit template therefore needs to specify a unique generic component id for this instantiation of the interface.

All subsequent components in a toolkit are defined as out of scope of the TPEG Application; i.e. the toolkit on its own defines subcomponents beginning with 0. With that on one hand application decoder must be aware that component ids of the application may be repeated in sub components of a toolkit. On the other hand further development of application and toolkit can be done independently.

6.5 Application design principles

This clause describes design principles that will be helpful in building TPEG applications. A fundamental assumption is that applications will develop and new features will be added. If design principles are adopted properly then older decoders will still operate properly after extending features. Correct design should permit applications to be upgraded and extended over time, providing new features to new decoders, and yet permit existing decoders to continue to operate.

6.5.1 Variable data structures

Switches may be included within an application, which permit variations in the subsequent data structure. However, the switch fixes the values of variations. A new type cannot be introduced without breaking backward compatibility. This may be achieved by using components. When new features are likely to be incorporated, attention should be given to the fact that old decoders just ‘skip over’ new data fields and still expect the old components if they were mandatory.

6.5.2 Re-usable and extendable structures

Within an application there will be data structures, which are used repeatedly in a variety of places. There will also certainly be an ever-growing set of structures, as the application protocol develops and incorporates new features. Component templates may be used to minimize the number of occasions within the decoder’s software in which the structure needs to be defined, and to permit an increasing variety of structures to be used in a given location.

6.5.3 Validity of declarative structures

The Identifier of a component is uniquely defined within each application. The same number may be used in different applications for completely different purposes. Within an application one identifier designates one definition of a component. The design of an application may use components to implement placeholders or to change the composition of elements in a fixed structure.

7 TPEG data stream description

7.1 Diagrammatic hierarchy representation of frame structure

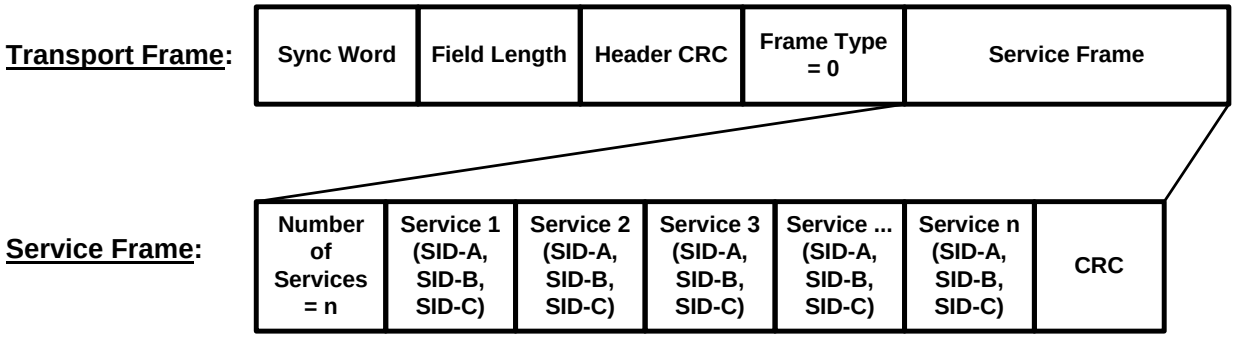


Figure 4 — TPEG Frame Structure, Frame Type = 0 (i.e. stream directory)

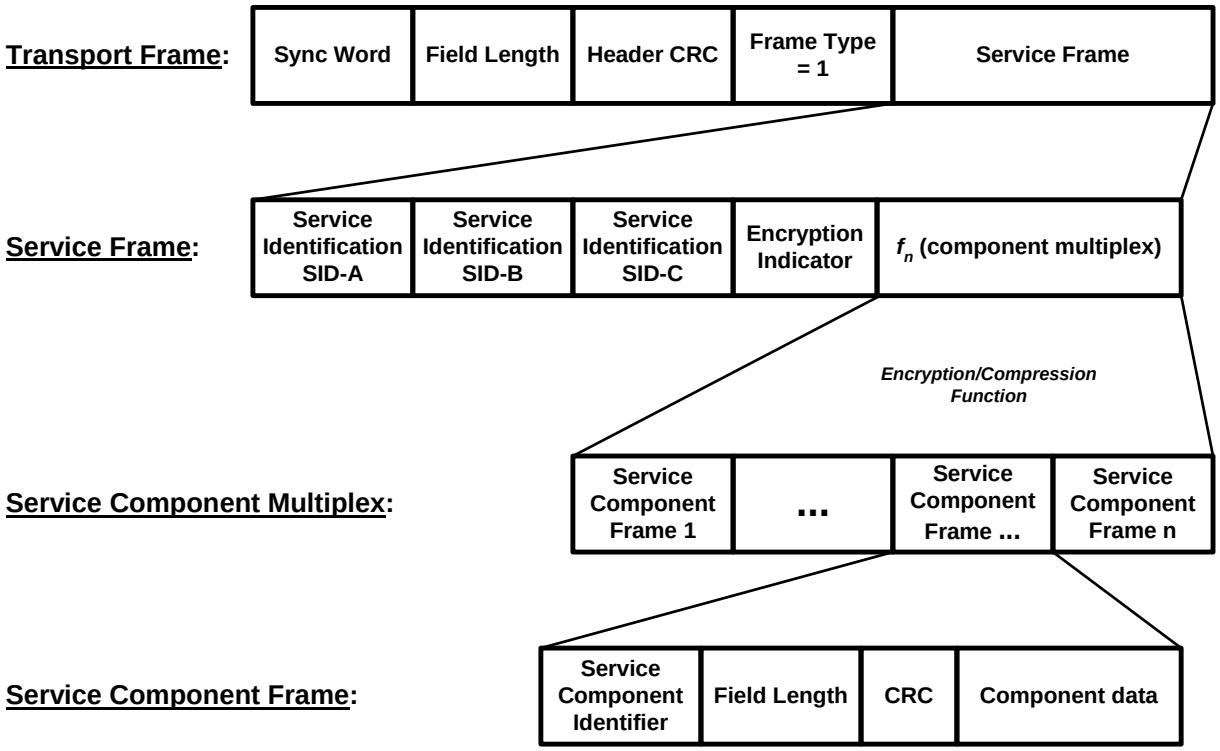


Figure 5 — TPEG Frame Structure, Frame Type = 1 (i.e. conventional data)

7.2 Syntactical Representation of the TPEG Stream

7.2.1 TPEG transport frame structure

The following boxes are the syntactical representation of the TPEG frame structure shown in Clause 7.1 The byte stream contains consecutive transport frames. Each frame includes:

The synchronization word (syncword)	2 bytes	(See Clause A.3.3.1)
The length of the service frame in bytes (field length)	2 bytes	(See Clause A.3.3.2)
The header CRC	2 bytes	(See Clause A.3.3.3)
The frame type indicator	1 byte	(See Clause A.3.3.4)
The service frame	n bytes (n = Field Length)	

The byte stream is built according to the above-mentioned repetitive structure of transport frames. Normally one transport frame should follow another directly, however if any spacing bytes are required these should be set to 0 hex (padding bytes).

<TpegStream>:=	: The data stream.
infinite {	: Control element, (loop continues infinitely)
n * **<IntUnTi>**(0)**,**	: Any number of padding bytes (0 hex)
<TransportFrame>	: Transport frames
};	

<TransportFrame>:=	
<IntUnLi>(FF0F hex)**,**	: Sync word (FF0F hex)
<IntUnLi>(m)**,**	: Number of bytes in Service Frame
<CRC>(headCRC)**,**	: Header CRC, (See Clause A.3.3.4)
<IntUnTi>(x),	: Frame type of service frame
<ServiceFrame(x)>;	: Any service frame follows

7.2.2 TPEG service frame template structure

This service frame comprises:

<ServiceFrame(x)>:=	: Template for service frame
n * **<byte>;**	: Content of service frame

7.2.3 Service frame of frame type = 0

The service frame is solely used to transport the stream directory information.

Number of services (n)	1 byte
n *(SID-A, SID-B, SID-C)	n * (3 bytes)
CRC	2 bytes

<StreamDirectory<ServiceFrame(0)>>:=	: Stream directory
<IntUnTi>(n)**,**	: Number of services
n * **<ServiceIdentifier>,**	: Any number of Service IDs
<CRC>;	: CRC of Service IDs

7.2.4 Service frame of frame type = 1

Each service frame comprises:

SID-A, SID-B, SID-C	3 bytes	(See Clause A.3.4.2)
The encryption indicator	1 byte	(See Clause A.3.4.1)
The component data	m bytes	

The service level is defined by the service frame. Each transport frame carries one and only one service frame. The service frame includes a component multiplex comprising one or more component frames.

Each service frame may contain a different range and number of component frames as required by the service provider.

Each transport frame may be used by only one service provider and one dedicated service, which supports a mixture of applications. A multiplex of service providers or services is realized by concatenation of multiple transport frames. Each service frame includes service information that comprises the service identification elements and the encryption indicator.

<ConventionalData<ServiceFrame(1)>>:=	: Conventional data
<ServiceIdentifier>,	: Service identification
<IntUnTi>(encIdentifier)**,**	: Encryption indicator n. 0 = no encryption
f_n**(<ServCompMultiplex>);**	: Function f_n (…) is utilized according to the chosen encryption algorithm

7.2.5 TPEG service component frame multiplex

The component multiplex is a collection of one or more component frames, the type and order of which are freely determined by the service provider. The resultant multiplex is transformed according to the encryption method required (if the encryption indicator is not 0) or is left unchanged (if the encryption indicator = 0). The length of the resultant data must be less than or equal to 65531 bytes.

<ServCompMultiplex>:=	
n * **<ServCompFrame>**(data)**;**	: Any number of any component frames

7.2.6 Interface to application specific frames

The service component frame introduces the application specific code. This means further details of the data stream are specified by the application specification. In the history for different needs slightly different frames have been defined in the existing application specifications. To harmonize this kind of frames, especially for new developments of specifications, this clause specifies not only a basic frame, which is required for any application but also a selection of possible other frames, whereof an application can just choose one without the need to specify its own frame.

An application specification, however, can specify its own frame, which shall at minimum include the following base service component frame as first sub type.

7.2.6.1 TPEG base service component frame structure

In a TPEG data stream it shall be possible to have not only one content stream but more; even different from the same application. This is possible with the help of the Service and Network Information (SNI) Application, which is served like variable directory information in the data stream. Therein a table defines a unique number for any content stream being transmitted. This includes also the definition which application is expected in one specific frame. In other words the frame starts not with a typical interface template, but with a header, defining three first values being in common with all service component frames. Therefore, any service component frame is built as shown below:

<**ServCompFrame**>:=	: Service component frame
<**ServCompFrameHeader**>(header),	: Common service component header
<**ApplicationData**>(data);	: Component data

Where the service component header is specified as:

<**ServCompFrameHeader**>:=	: Common service component frame header
<**IntUnTi**>(scId),	: Service component identifier (scid is defined by SNI service component designating the application in this service component frame)
<**IntUnLi**>(lengthInByte),	: Length, n, of component data in bytes
<**CRC**>(headerCRC);	: Header CRC (See Clause A.3.5.3)

At the component level data is carried in component frames which have a limited length. If applications require greater capacity then the application must be designed to distribute data between component frames and to recombine this information in the decoder.

The inclusion of the field length enables the decoder to skip a component.

The maximum field length of the component data (assuming that there is no transformation, and only one component is included in the service frame) = 65526.

7.2.6.2 TPEG specialized service component data schemata

It is in interest of consistency to make sure that service component frames still become defined in as similar as possible in different applications. Specifically with three further attributes being of general nature. The following proposed specialized service component data schemata can be used to inform on this general level about following information:

a) The application data of a component frame with **dataCRC** is error-free.

Data CRC on this level makes it possible, that in case of errors only the service component frame (e.g. one relatively small package of data) would be lost. Other parts of the service multiplex may still be valid and could still be used. (See Clause A.3.5.4)

b) Count of messages the service component frame contains named **messageCount**.

Sometimes it is useful not only to know the opaque count of bytes, but also how many different message have to be expected by the decoder (e.g. for displaying purpose).

c) Prioritization can be made by assigning a **groupPriority**.

In some cases the different service components received shall not just be handled by a FIFO buffer but also with some qualification of priority of messages. In this case high priority message may take precedence over other messages in the decoder. These may be presented to the user even before low priority messages are decoded.

7.2.6.2.1 Service component data with dataCRC

Any application should at least specify a data CRC as defined in Clause A.3.5.4 at the end of application data ensuring that bit errors can be detected on service component frame level.

< **ServCompFrameProtected** >:=	: CRC protected service component frame
<**ServCompFrameHeader**>(header),	: Component frame header as defined in A.3.2.6.1
external <**ApplicationContent>**(content)**,**	: Content specified by the individual application
<**CRC**>(dataCRC);	: CRC starting with first byte after the header

7.2.6.2.2 Service component data with dataCRC and messageCount

This service frame is used for applications containing messages more or less directly presented to the user which indicate already on frame level how many messages are to be expected. Data CRC is contained as well.

< **ServCompFrameCountedProtected**>:=	: CRC protected service component frame with message count
<**ServCompFrameHeader**>(header),	: Component frame header as defined in A.3.2.6.1
<**IntUnTi**>(messageCount),	: count of messages in this ApplicationContent
external <**ApplicationContent**>(content),	: actual payload of the application
<**CRC**>(dataCRC);	: CRC starting with first byte after the header

7.2.6.2.3 Service component data with dataCRC and groupPriority

When messages need to be grouped by priority, this service component frame is used. If not all messages within the frame have the same priority, 'typ007_000: undefined' shall be used. Data CRC is contained as well.

< **ServCompFramePrioritisedProtected**>:=	: CRC protected service component frame with message count
<**ServCompFrameHeader**>(header),	: Component frame header as defined in A.3.2.6.1
<**typ007:Priority**>(groupPriority),	: group priority applicable to all messages in this ApplicationContent
external <**ApplicationContent**>(content),	: actual payload of the application
<**CRC**>(dataCRC);	: CRC starting with first byte of after the header

7.2.6.2.4 Service component frame with dataCRC, groupPriority, and messageCount

Additionally, an application can also make use of all features described in previous clauses.

< **ServCompFramePrioritisedCountedProtected**>:=	: CRC protected service component frame with group priority and message count
<**ServCompFrameHeader**>(header),	: Component frame header as defined in A.3.2.6.
<**typ007:Priority**>(groupPriority),	: group priority applicable to all messages in the ApplicationContent
<**IntUnTi**>(messageCount),	: count of messages in this ApplicationContent
external <**ApplicationContent**>(content),	: actual payload of the application
<**CRC**>(dataCRC);	: CRC starting with first byte after the header

7.2.6.3 Example of an application implementing a service component frame

An application specification is required to specify first the component frame just as a written sentence. It may for information repeat the definition of the frame, but in this case it shall add a note, that this definition can be superseded by a future release of this specification.

As second definition tree of application starts with:

<**ApplicationContent**>:=	: link provided by SSF
n * <**MyComponent**>(comp);	: n root components of the application

<MyComponent<Component(0)>>:=	
<IntUnTi>(0),	: id = 1
<IntUnLoMB>(compLengthInByte),	: length of the component in bytes
<IntUnLoMB>(attributeBlockLengthInByte),	: length of the attribute block in bytes
<ShortString>(myText),	: some first attribute of the application
<SubComp>(sub);	: some sub components of Component(0)

7.3 Description of data on Transport level

7.3.1 Syncword

The syncword is 2 bytes long, and has the value of FF0F hex.

The nibbles F hex and 0 hex have been chosen for simplicity of processing in decoders. The patterns 0000 hex and FFFF hex were deprecated to avoid the probability of false triggering in the cases of some commonly used transmission channels.

7.3.2 Field length

The field length consists of 2 bytes and represents the number of bytes in the service frame.

This derives from the need of variable length frames.

7.3.3 Header CRC

The Header CRC is two bytes long, and is based on the ITU-T polynomial $x^{16} + x^{12} + x^5 + 1$. The Header CRC is calculated on 16 bytes including the syncword, the field length, the frame type and the first 11 bytes of the service frame. In the case that a service frame is shorter than 11 bytes, the sync word, the field length, the frame type and the *whole* service frame shall be taken into account.

In this case the Header CRC calculation does not run into the next transport frame.

The calculation of the CRC is described in Annex C.

7.3.4 Frame type

The frame type (FTY) indicates the content of the service frame. Its length is 1 byte. The following table gives the meaning of the frame type:

FTY value (dec):	Content of service frame:	Kind of information in service frame:
0	Number of services, n * (SID-A, SID-B, SID-C)	Stream directory information
1	SID-A, SID-B, SID-C, Encryption ID, Component Multiplex	Conventional service frame data

If FTY = 0, an extra CRC calculation is done over the whole service frame, i.e. starting with n (number of services) and ending with the last SID-C of the last service.

The calculation of the CRC is described in Annex C.

7.3.5 Synchronization method

A three-step synchronization algorithm can be implemented to synchronize the receiver:

a) search for an FF0F hex value;

b) calculate and check the header CRC, which follows;

c) check the two bytes, which follow the end of the service frame as defined by the field length.

The two bytes following the end of the service frame should either be a sync word or 00 hex, when spaces are inserted.

7.3.6 Error detection

The CRC header provides error detection and protection for the synchronization elements and not for the data within the service frame (except the first 11 bytes, when applicable).

7.4 Description of data on Service level

7.4.1 Encryption indicator

Length: 1 byte

The encryption indicator is defined as one byte according to TPEG primitive syntax. If the indicator has value 00 hex all data in the component multiplex are non-encrypted. Every other value of the encryption indicator indicates that one of several mechanisms for data encryption or compression has been utilized for all data in the following data multiplex. The encryption/compression technique and algorithms may be freely chosen by the service provider.

0	=	no encryption/compression
1 to 127	=	reserved for standardized methods
128 to 255	=	may be freely used by each service provider, may indicate the use of proprietary methods

7.4.2 Service identification

The service IDs are structured in a similar way to Internet IP addresses as follows:

SID-A . SID-B . SID-C

The combination of these three SID elements must be uniquely allocated on a worldwide basis.

The following address allocation system applies:

— SID range for TPEG technical tests SIDs = 000.000.000 - 000.127.255

— SID range for TPEG public tests SIDs = 000.128.000 - 000.255.255

— SID range for TPEG regular public services SIDs = 001.000.000 - 100.255.255

— SID range: reserved for future use SIDs = 101.000.000 - 255.255.255

NOTE The above allocations and structure is significantly changed from that originally specified in CEN ISO/TS 18234-2.

7.5 Description of data on Service component level

7.5.1 Service component identifier

The service component identifier with the value 0 is reserved for the SNI Application. (See CEN ISO/TS 18234-3)

7.5.2 Field length

The field length consists of 2 bytes and represents the number of bytes of the component data.

7.5.3 Service component frame header CRC

The component header CRC is two bytes long, and based on the ITU-T polynomial $x^{16}+x^{12}+x^{5}+1$.

The component header CRC is calculated from the service component identifier, the field length and the first 13 bytes of the component data. In the case of component data shorter than 13 bytes, the component identifier, the field length and all component data shall be taken into account.

The calculation of the CRC is described in Annex C.

7.5.4 Service component frame data CRC

The DataCRC is two bytes long, and is based on the ITU polynomial $x^{16}+x^{12}+x^{5}+1$. This CRC is calculated from all the bytes of the service component frame data after the service component frame header.

The calculation of the CRC is described in Annex C.

Annex A
(normative)

Character tables

A.1 Character tables

The default character coding table used in TPEG is ISO/IEC 8859-1.

A.2 Reference character table index

Table A.1 — Reference character table index

t = Char-Tab	k = bytes/char	Name of Character Table
0	-	Reserved
1	1	ISO/IEC 8859-1 (Default)
2	1	ISO/IEC 8859-2
3	1	ISO/IEC 8859-3
4	1	ISO/IEC 8859-4
5	1	ISO/IEC 8859-5
6	1	ISO/IEC 8859-6
7	1	ISO/IEC 8859-7
8	1	ISO/IEC 8859-8
9	1	ISO/IEC 8859-9
10	1	ISO/IEC 8859-10
11	1	Reserved
12	1	Reserved
13	1	ISO/IEC 8859-13
14	1	ISO/IEC 8859-14
15	1	ISO/IEC 8859-15
....	....	....
125	1	Unicode ISO/IEC 10646 UTF-8
126	2	Unicode ISO/IEC 10646 UTF-16
127	4	Unicode ISO/IEC 10646 UTF-32
128		Reserved
...	....	....
255		Reserved

The selection of TPEG coding tables is implemented according to the character table switch in 6.3.1.4 with the following value ranges:

a) The range t = 1 to 127 is reserved for standardized character tables.

b) The range t = 128 to 255 may be freely used by a service provider and may indicate the use of proprietary character tables. In combination with the service provider identification this guarantees uniqueness.

Annex B
(normative)

Method for coding quantities of objects

B.1 Numag derivation

Within applications of TPEG there is a frequent need to describe with a single byte a quantity of people, objects, etc., using a non-linear coding system which provides a high resolution for low numbers and progressively lower resolution for higher numbers.

The primitive **<numag>** describes, in a single byte, quantities which lie in the range 0 - 3 000 000.

<numag>:=	: Counting numbers with magnitude, $0 <= r <= 3 \times 10^6$
<intunti>(n)**;**	: Where $r := (5 + \text{sign}(n-5) \times (\text{abs}(n-5) \bmod 45)) \times 10^{(n-5) \text{ div } 45}$

The following formula translates the value, n, to the result, r.

$r := (5 + \text{sign}(n-5) \times (\text{abs}(n-5) \bmod 45)) \times 10^{(n-5) \text{ div } 45}$

This formula, which produces the sequence of numbers shown in Table B.2, is calculated as follows:

a) n is an integer in the range 0.255 and is used to code the number, r

b) Intermediate values are generated:

$a := \text{sign}(n-5)$

$b := \text{abs}(n-5) \bmod 45$

$c := (n-5) \text{ div } 45$

c) The result, r, is generated from these intermediate values as follows:

$r := (a \times b + 5) \times 10^c$

B.2 Numag table

Table B.1 — Numag table

n:	R:	n:	r:	n:	r:	n:	r:
0	0	64	190	128	3 800	192	120 000
1	1	65	200	129	3 900	193	130 000
2	2	66	210	130	4 000	194	140 000
3	3	67	220	131	4 100	195	150 000
4	4	68	230	132	4 200	196	160 000
5	5	69	240	133	4 300	197	170 000
6	6	70	250	134	4 400	198	180 000
7	7	71	260	135	4 500	199	190 000
8	8	72	270	136	4 600	200	200 000
9	9	73	280	137	4 700	201	210 000
10	10	74	290	138	4 800	202	220 000
11	11	75	300	139	4 900	203	230 000
12	12	76	310	140	5 000	204	240 000
13	13	77	320	141	6 000	205	250 000
14	14	78	330	142	7 000	206	260 000
15	15	79	340	143	8 000	207	270 000
16	16	80	350	144	9 000	208	280 000
17	17	81	360	145	10 000	209	290 000
18	18	82	370	146	11 000	210	300 000
19	19	83	380	147	12 000	211	310 000
20	20	84	390	148	13 000	212	320 000
21	21	85	400	149	14 000	213	330 000
22	22	86	410	150	15 000	214	340 000
23	23	87	420	151	16 000	215	350 000
24	24	88	430	152	17 000	216	360 000
25	25	89	440	153	18 000	217	370 000
26	26	90	450	154	19 000	218	380 000
27	27	91	460	155	20 000	219	390 000
28	28	92	470	156	21 000	220	400 000
29	29	93	480	157	22 000	221	410 000
30	30	94	490	158	23 000	222	420 000
31	31	95	500	159	24 000	223	430 000
32	32	96	600	160	25 000	224	440 000
33	33	97	700	161	26 000	225	450 000
34	34	98	800	162	27 000	226	460 000
35	35	99	900	163	28 000	227	470 000
36	36	100	1 000	164	29 000	228	480 000
37	37	101	1 100	165	30 000	229	490 000
38	38	102	1 200	166	31 000	230	500 000
39	39	103	1 300	167	32 000	231	600 000
40	40	104	1 400	168	33 000	232	700 000
41	41	105	1 500	169	34 000	233	800 000
42	42	106	1 600	170	35 000	234	900 000
43	43	107	1 700	171	36 000	235	1 000 000
44	44	108	1 800	172	37 000	236	1 100 000
45	45	109	1 900	173	38 000	237	1 200 000
46	46	110	2 000	174	39 000	238	1 300 000
47	47	111	2 100	175	40 000	239	1 400 000
48	48	112	2 200	176	41 000	240	1 500 000
49	49	113	2 300	177	42 000	241	1 600 000
50	50	114	2 400	178	43 000	242	1 700 000
51	60	115	2 500	179	44 000	243	1 800 000
52	70	116	2 600	180	45 000	244	1 900 000
53	80	117	2 700	181	46 000	245	2 000 000
54	90	118	2 800	182	47 000	246	2 100 000
55	100	119	2 900	183	48 000	247	2 200 000
56	110	120	3 000	184	49 000	248	2 300 000
57	120	121	3 100	185	50 000	249	2 400 000
58	130	122	3 200	186	60 000	250	2 500 000
59	140	123	3 300	187	70 000	251	2 600 000
60	150	124	3 400	188	80 000	252	2 700 000
61	160	125	3 500	189	90 000	253	2 800 000
62	170	126	3 600	190	100 000	254	2 900 000
63	180	127	3 700	191	110 000	255	3 000 000

Annex C
(normative)

CRC calculation

C.1 CRC calculation

The TPEG **<crc>** primitive is represented by a word **<intunli>** which itself represents the result of a 16-bit cyclic redundancy check (CRC) calculation upon a designated range of elements.

The calculation starts with the most significant bit of the first designated element field and ends with the least significant bit of the last byte of the last designated element.

The divisor polynomial used to generate the CRC is:

$$x^{16} + x^{12} + x^{5} + 1$$

The CRC is initialized by a value of FFFF hex, and the two check bytes are formed from the inverse of the result (1's complement). The eight most significant bits are represented by the first check field byte, and the eight least significant bits are represented by the last check field byte.

Example: When applied to a sequence of 47 bytes:

32 44 31 31 31 32 33 34 30 31 30 31 30 35 41 42 43 44 31 32 33 46 30 58 58 58 58 31 31 30 36 39 32 31 32 34 39 31 30 30 30 33 32 30 30 36 36 hex,

the CRC generated is 97 23 hex.

C.2 ITU-T (formerly CCITT) CRC calculation in PASCAL

Type STRING is a PACKED ARRAY of CHAR with zero'th element holding the length of the string.

SWAP is a library function that swaps the high- and low-order bytes of the argument.

EXAMPLE 1

```
VAR X: WORD;
BEGIN
 X:= SWAP ($1234)        [$3412]
END;
```

LO is a library function which returns the low-order byte of the argument.

EXAMPLE 2

```
VAR W: WORD;
BEGIN
 W:= LO ($1234)      [$34]
END;

FUNCTION CRCVALUE (STRINGTOEVAL : STRING): INTEGER;
VAR
COUNT: BYTE;
TEMPCRC: WORD;
BEGIN
 TEMPCRC:= $FFFF;
 FOR COUNT:= 1 TO LENGTH (STRINGTOEVAL) DO
```

```
        BEGIN
            TEMPCRC:= SWAP (TEMPCRC) XOR ORD (STRINGTOEVAL [COUNT]);
            TEMPCRC:= TEMPCRC XOR (LO (TEMPCRC) SHR 4);
            TEMPCRC:= TEMPCRC XOR (SWAP (LO (TEMPCRC)) SHL 4) XOR (LO (TEMPCRC) SHL 5)
        END;
        CRCVALUE:= TEMPCRC XOR $FFFF
       END; [OF FUNCTION CRCVALUE]
```

C.3 ITU-T (formerly CCITT) CRC calculation in C notation

```
#define swap(a) (((a)<<8)|((a)>>8))
//------------------------------------------------------------------------------
unsigned short usCalculCRC(unsigned char *buf,unsigned long lg)
//------------------------------------------------------------------------------
{
unsigned short crc;
unsigned long count;
crc= 0xFFFF;
for (count= 0; count < lg; count++)
    {
    crc = (unsigned short) (swap(crc) ^ (unsigned short)buf[count]);
    crc ^= ((unsigned char)(crc) >> 4);
    crc = (unsigned short) (crc ^ (swap((unsigned char)(crc)) << 4)
            ^ ((unsigned char)(crc) << 5));
    }

return((unsigned short)(crc ^ 0xFFFF));
}
```

Annex D
(normative)

Time calculation

D.1 Time calculation

The TPEG <time_t> compound element is represented by the primitive element <intunlo>. It represents the number of seconds since 1970-01-01T00:00:00 Universal Coordinated Time (UTC). Since <intunlo> ranges from 0..4 294 967 295, Date and Time ranging from 1970-01-01T00:00:00 to 2106-02-07T06:28:15 can be represented.

This annex provides some functions to code from date and time to seconds and vice versa. It just intends to provide one example of how to implement time calculation for TPEG. Other and probably better solutions will exist.

Some examples of calculation have been provided to check the time calculation. The examples specified in the table are well chosen based on highly possible bugs (range, representation, leap year, Y2K problem, signed/unsigned overflow, etc.).

Table D.1 — Examples of time calculation

Seconds (decimal)	Seconds (hexadecimal)	Date/Time
0	0x00000000	1970-01-01T00:00:00Z
1500	0x000005DC	1970-01-01T00:25:00Z
2429884	0x002513BC	1970-01-29T02:58:04Z
68179407	0x041055CF	1972-02-29T02:43:27Z
946684800	0x386D4380	2000-01-01T00:00:00Z
951788609	0x38BB2441	2000-02-29T01:43:29Z
970315500	0x39D5D6EC	2000-09-30T12:05:00Z
1102118400	0x41B0FE00	2004-12-04T00:00:00Z
2147483646	0x7FFFFFFE	2038-01-19T03:14:06Z
2147483648	0x80000000	2038-01-19T03:14:08Z
4107580093	0xF4D4B2BD	2100-03-01T10:28:13Z
4294967295	0xFFFFFFFF	2106-02-07T06:28:15Z

D.2 Time calculation in C notation

TPEG time is calculated according the following formulas:

```
#define SECS_PER_MIN    60
#define MINS_PER_HOUR   60
#define HOURS_PER_DAY   24
#define DAYS_PER_WEEK   7
#define DAYS_PER_NYEAR  365
#define DAYS_PER_LYEAR  366
#define SECS_PER_HOUR   (SECS_PER_MIN * MINS_PER_HOUR)
#define SECS_PER_DAY    (SECS_PER_HOUR * HOURS_PER_DAY)
#define MONS_PER_YEAR   12

#define TM_SUNDAY   0
#define TM_MONDAY   1
#define TM_TUESDAY  2
#define TM_WEDNESDAY    3
#define TM_THURSDAY 4
```

```
#define TM_FRIDAY   5
#define TM_SATURDAY 6

#define TM_JANUARY  0
#define TM_FEBRUARY 1
#define TM_MARCH    2
#define TM_APRIL    3
#define TM_MAY  4
#define TM_JUNE 5
#define TM_JULY 6
#define TM_AUGUST   7
#define TM_SEPTEMBER    8
#define TM_OCTOBER  9
#define TM_NOVEMBER 10
#define TM_DECEMBER 11

#define L_JANUARY   31
#define L_FEBRUARY  28
#define L_LONG_FEBRUARY 29
#define L_MARCH 31
#define L_APRIL 30
#define L_MAY   31
#define L_JUNE  30
#define L_JULY  31
#define L_AUGUST    31
#define L_SEPTEMBER 30
#define L_OCTOBER   31
#define L_NOVEMBER  30
#define L_DECEMBER  31

#define TM_YEAR_BASE    1900

#define EPOCH_YEAR  1970
#define EPOCH_WDAY  TM_THURSDAY

#define isleap(y)   (((((y) % 4) == 0) && (((y) % 100) != 0)) || (((y) % 400) == 0))
/* Leap-year: If year divided without remainder by 4, but not by 100.
 * Exception: also leap-year if divided by 400 */

/* macro to get the number of days in a year (depends on being a leap year) */
#define days_in_year(y) (isleap((y)) ? DAYS_PER_LYEAR : DAYS_PER_NYEAR)

/* take days in month from monthdays except for leap year February */
#define days_in_month(y,m)  ((isleap((y)) && ((m) == TM_FEBRUARY)) \
    ? L_LONG_FEBRUARY : monthdays[(m)])

typedef struct {
    int tm_sec; /* 0 - 59 */
    int tm_min; /* 0 - 59 */
    int tm_hour;    /* 0 - 23 */
    int tm_mday;    /* 1 - 31 */
    int tm_mon; /* 0 - 11 */
    int tm_year;    /* years since 1900 */
    int tm_wday;    /* 0 - 6 sunday=0 */
    int tm_yday;    /* 0 - 365 */
} tm;

static int monthdays[MONS_PER_YEAR] = {
    L_JANUARY, L_FEBRUARY, L_MARCH, L_APRIL, L_MAY, L_JUNE, L_JULY,
    L_AUGUST, L_SEPTEMBER, L_OCTOBER, L_NOVEMBER, L_DECEMBER
};

static void
calc_time(unsigned long timep, tm *tmp)
{
    unsigned long   days;
    unsigned long   rem;
    int y;

    days = timep / SECS_PER_DAY;
    rem = timep % SECS_PER_DAY;
    tmp->tm_hour = (int) (rem / SECS_PER_HOUR);
    rem = rem % SECS_PER_HOUR;
    tmp->tm_min = (int) (rem / SECS_PER_MIN);
    tmp->tm_sec = (int) (rem % SECS_PER_MIN);
    tmp->tm_wday = (int) ((EPOCH_WDAY + days) % DAYS_PER_WEEK);
```

```
    for (y = EPOCH_YEAR; days >= days_in_year(y); y++) {
        days -= days_in_year(y);
    }

    for (tmp->tm_mon = TM_JANUARY;
        days >= days_in_month(y, tmp->tm_mon);
        (tmp->tm_mon)++) {
        days -= days_in_month(y, tmp->tm_mon);
    }

    tmp->tm_year = y - TM_YEAR_BASE;
    tmp->tm_yday = (int) days;
    tmp->tm_mday = (int) (days + 1);
}

static unsigned long
recalc_time(int year, int month, int day, int hour, int min, int sec)
/* year : 1970-2106, month 1-12, day 1-31, hour 0-23, min 0-59, sec 0-59 */
{
    unsigned long   secs;
    int days;
    int y;
    int m;

    days = 0;

    for (y = year - 1; y >= EPOCH_YEAR; y--) {
        days += days_in_year(y);
    }

    for (m = TM_JANUARY; m < (month - 1); m++) {
        days += days_in_month(year, m);
    }

    days += day - 1;

    secs = (days * SECS_PER_DAY);
    secs += (hour * SECS_PER_HOUR);
    secs += (min * SECS_PER_MIN);
    secs += sec;

    return (secs);
}
```

Annex E
(informative)

A description of the TPEG byte-stream using C-type notation

E.1 Explanation

A byte stream consists of a sequence of data elements. Each data element in the byte stream is in bold type. It is described by its name, its length in bytes and a mnemonic for its type. A sequence of data elements may be defined as a function. The scope of the name of the data elements is local to the function. To address a data element of a parent function, the dotted notation, 'function**.**data_element_name' is used. This notation can be nested to work back to the top-level function, e.g. function1**.**function2**.**data_element_name.

The action caused by a decoded data element in a byte stream depends on the value of that data element and on data elements previously decoded. The decoding of the data elements and definition of the state variables used in their decoding are described in the clauses containing the semantic description of the syntax. The following constructs are used to express the conditions when data elements are present and are in normal type:

E.2 Definition of data elements

data_element [ ]	data_element [ ] is an array of data. The number of data elements is indicated by the context
data_element [n]	data_element [n] is the n+1 th element of an array of data
data_element [m][n]	data_element [m][n] is the m+1, n+1 th element of a two-dimensional array of data
data_element [m..n]	is the inclusive range of bits between bit m and bit n in the data_element

The syntax described in procedural terms defines a correct and error-free input bitstream. Actual decoders must include a means to look for start codes and sync bytes in order to begin decoding correctly and to identify errors, erasures and insertions while decoding. The methods to identify these situations and the actions to be taken are not standardized.

E.3 Definition of conditional expressions

NOTE This syntax uses the 'C'-code convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true.

while (condition) {
data_element
…
}
	If the condition is true, the group of data elements occurs next in the data-stream. This repeats until the condition is not true.

do {
data_element
…
}
while (condition)
	The data element always occurs at least once. The data element is repeated until the condition is not true.

if (condition) {
data_element
…
}
else {
data_element
…
}
	If the condition is true, then the first group of data elements occurs next in the data-stream.
	
	If the condition is not true, then the second group of data elements occurs next in the data-stream.

for (i=0 ; i<n ; i++) {
data_element
…
}
	The group of data elements occurs n times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to zero for the first occurrence, incremented to 1 for the second occurrence and so forth.

E.4 Byte-stream representation of the TPEG hierarchy

E.4.1 Definition of nextbyte function

The function nextbyte() permits comparison of a byte string with the next bytes to be decoded in the byte stream. The function does not consume any bytes from the stream.

E.4.2 Definition of next_start_code function

The next_start_code() function removes any zero byte stuffing and locates the next start code.

Syntax	No. of bytes	Mnemonic

next_start_code() {
while (TRUE) {
while (nextbyte() != 'FF hex') {
byte;
}
byte;
if (nextbyte() == '0F hex') {
byte;
return;
}
}
}
	
	1
	
	1
	
	1
		
		garbage
		
		FF hex
		
		0F hex

E.4.3 Definition of tpeg_stream function

The tpeg_stream() function processes the input byte stream.

Syntax	No. of bytes	Mnemonic
tpeg_stream() {		
while (TRUE) {		
next_start_code();		
field_length;	**2**	**intunli**
header_crc;	**2**	**crc**
frame_type;	**1**	**fty**
if (header_crc_check()) {		
service_frame(frame_type);		
}		
}		
}		

Syntax	No. of bytes	Mnemonic
service_frame(00) {		
number_of_services;	**1**	**intunti**
while (number_of_services > 0) {		
service_identification_a;	**1**	**intunti**
service_identification_b;	**1**	**intunti**
service_identification_c;	**1**	**intunti**
number_of_services = number_of_services - 1;		
}		
service_crc;	**2**	**crc**
}		

Syntax	No. of bytes	Mnemonic
service_frame(01) {		
service_identification_a;	**1**	**intunti**
service_identification_b;	**1**	**intunti**
service_identification_c;	**1**	**intunti**
encryption_indicator;	**1**	**intunti**
component_multiplex();		
}		

Syntax	No. of bytes	Mnemonic
component_multiplex() {		
if (encryption_indicator == 0) {		
length = tf_field_length - 5;		
do {		
component_identifier;	**1**	**intunti**
field_length;	**2**	**intunli**
component_header_crc;	**2**	**crc**
if (component_header_crc_check()) {		
component_data();		
}		
} while (length = length - (field_length + 5));		
} else {		
decompression();		
get_new_length_and_add_five();		
encryption_indicator = 0;		
component_multiplex();		
}		
}		

ICS 03.220.01; 35.240.60

Price based on 34 pages