INTERNATIONAL STANDARD

ISO/IEC 14496-1

Fourth edition
2010-06-01

Information technology — Coding of audio-visual objects —

Part 1:
Systems

Technologies de l'information — Codage des objets audiovisuels —

Partie 1: Systèmes

Reference number
ISO/IEC 14496-1:2010(E)

Contents

Page

Foreword iv
0 Introduction vi
1 Scope 1
2 Normative references 1
3 Additional references 2
4 Terms and definitions 2
5 Abbreviated terms 10
6 Conventions 11
7 Streaming Framework 11
8 Syntactic Description Language 99
9 Profiles 110
Annex A (informative) Time Base Reconstruction 112
Annex B (informative) Registration procedure 115
Annex C (informative) The QoS Management Model for ISO/IEC 14496 Content 119
Annex D (informative) Conversion Between Time and Date Conventions 120
Annex E (informative) Graphical Representation of Object Descriptor and Sync Layer Syntax 122
Annex F (informative) Elementary Stream Interface 130
Annex G (informative) Upstream Walkthrough 132
Annex H (informative) Scene and Object Description Carrousel 137
Annex I (normative) Usage of ITU-T Recommendation H.264 | ISO/IEC 14496-10 AVC 138
Annex J (informative) Patent statements 141
Bibliography 144

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

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

ISO/IEC 14496-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This fourth edition cancels and replaces the third edition (ISO/IEC 14496-1:2004), which has been technically revised. It also incorporates the Amendments ISO/IEC 14496-1:2004/Amd.1:2005, ISO/IEC 14496-1:2004/Amd.2:2007, ISO/IEC 14496-1:2004/Amd.3:2007 and Technical Corrigenda ISO/IEC 14496-1:2004/Cor.1:2006 and ISO/IEC 14496-1:2004/Cor.2:2007.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects*

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description*

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

— *Part 16: Animation Framework eXtension (AFX)*

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

— *Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)*

— *Part 21: MPEG-J Graphics Framework eXtensions (GFX)*

— *Part 22: Open Font Format*

— *Part 23: Symbolic Music Representation*

— *Part 24: Audio and systems interaction*

— *Part 25: 3D Graphics Compression Model*

— *Part 26: Audio conformance*

— *Part 27: 3D Graphics conformance*

0 Introduction

0.1 Overview

ISO/IEC 14496 specifies a system for the communication of interactive audio-visual scenes. This specification includes the following elements.

a) The coded representation of natural or synthetic, two-dimensional (2D) or three-dimensional (3D) objects that can be manifested audibly and/or visually (audio-visual objects) (specified in Parts 2, 3, 10, 11, 16, 19, 20, 23 and 25 of ISO/IEC 14496).

b) The coded representation of the spatio-temporal positioning of audio-visual objects as well as their behavior in response to interaction (scene description, specified in Parts 11 and 20 of ISO/IEC 14496).

c) The coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content, specified in this Part and in Part 24 of ISO/IEC 14496).

d) A generic interface to the data stream delivery layer functionality (specified in Part 6 of ISO/IEC 14496).

e) An application engine for programmatic control of the player: format, delivery of downloadable Java byte code as well as its execution lifecycle and behavior through APIs (specified in Parts 11 and 21 of ISO/IEC 14496).

f) A file format to contain the media information of an ISO/IEC 14496 presentation in a flexible, extensible format to facilitate interchange, management, editing, and presentation of the media specified in Part 12 (ISO File Format), Part 14 (MP4 File Format) and Part 15 (AVC File Format) of ISO/IEC 14496.

g) The coded representation of font data and of information related to the management of text streams and font data streams (specified in Parts 17, 18 and 22 of ISO/IEC 14496).

The overall operation of a system communicating audio-visual scenes can be paraphrased as follows:

At the sending terminal, the audio-visual scene information is compressed, supplemented with synchronization information and passed to a delivery layer that multiplexes it into one or more coded binary streams that are transmitted or stored. At the receiving terminal, these streams are demultiplexed and decompressed. The audio-visual objects are composed according to the scene description and synchronization information and presented to the end user. The end user may have the option to interact with this presentation. Interaction information can be processed locally or transmitted back to the sending terminal. ISO/IEC 14496 defines the syntax and semantics of the bitstreams that convey such scene information, as well as the details of their decoding processes.

This part of ISO/IEC 14496 specifies the following tools.

— A terminal model for time and buffer management.

— A coded representation of metadata for the identification, description and logical dependencies of the elementary streams (object descriptors and other descriptors).

— A coded representation of descriptive audio-visual content information [object content information (OCI)].

— An interface to intellectual property management and protection (IPMP) systems.

— A coded representation of synchronization information (sync layer – SL).

— A multiplexed representation of individual elementary streams in a single stream (M4Mux).

These various elements are described functionally in this clause and specified in the normative clauses that follow.

0.2 Architecture

The information representation specified in ISO/IEC 14496 describes the means to create an interactive audio-visual scene in terms of coded audio-visual information and associated scene description information. The entity that composes and sends, or receives and presents such a coded representation of an interactive audio-visual scene is generically referred to as an “audio-visual terminal” or just “terminal”. This terminal may correspond to a stand-alone application or be part of an application system.

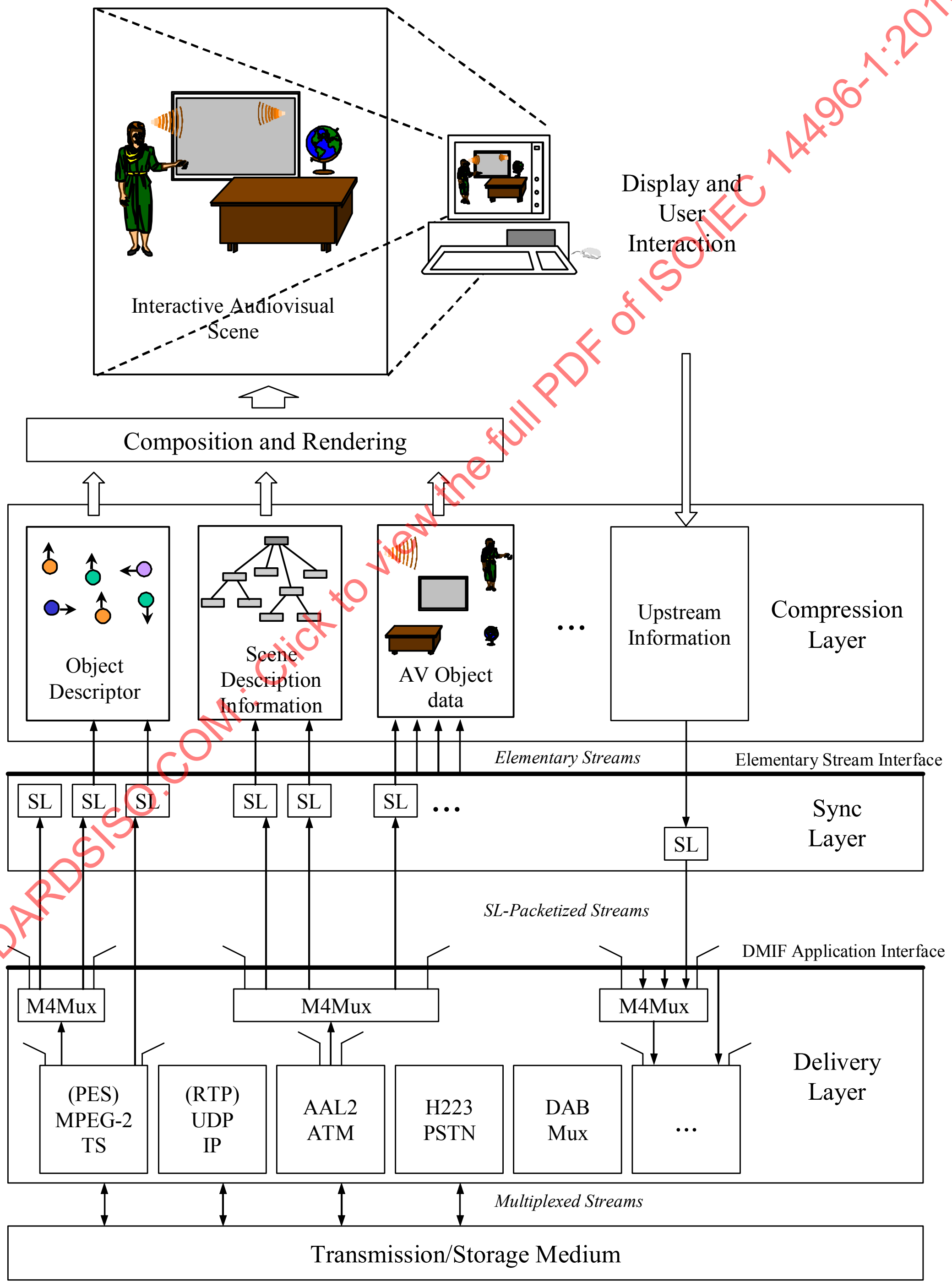


Figure 1 — The ISO/IEC 14496 Terminal Architecture

The basic operations performed by such a receiver terminal are as follows. Information that allows access to content complying with ISO/IEC 14496 is provided as initial session set up information to the terminal. Part 6 of ISO/IEC 14496 defines the procedures for establishing such session contexts as well as the interface to the delivery layer that generically abstracts the storage or transport medium. The initial set up information allows, in a recursive manner, to locate one or more elementary streams that are part of the coded content representation. Some of these elementary streams may be grouped together using the multiplexing tool described in ISO/IEC 14496-1.

Elementary streams contain the coded representation of either audio or visual data or scene description information or user interaction data or text or font data. Elementary streams may as well themselves convey information to identify streams, to describe logical dependencies between streams, or to describe information related to the content of the streams. Each elementary stream contains only one type of data.

Elementary streams are decoded using their respective stream-specific decoders. The audio-visual objects are composed according to the scene description information and presented by the terminal's presentation device(s). All these processes are synchronized according to the systems decoder model (SDM) using the synchronization information provided at the synchronization layer.

These basic operations are depicted in Figure 1, and are described in more detail below.

0.3 Terminal Model: Systems Decoder Model

The systems decoder model provides an abstract view of the behavior of a terminal complying with ISO/IEC 14496-1. Its purpose is to enable a sending terminal to predict how the receiving terminal will behave in terms of buffer management and synchronization when reconstructing the audio-visual information that comprises the presentation. The systems decoder model includes a systems timing model and a systems buffer model which are described briefly in the following Subclauses.

0.3.1 Timing Model

The timing model defines the mechanisms through which a receiving terminal establishes a notion of time that enables it to process time-dependent events. This model also allows the receiving terminal to establish mechanisms to maintain synchronization both across and within particular audio-visual objects as well as with user interaction events. In order to facilitate these functions at the receiving terminal, the timing model requires that the transmitted data streams contain implicit or explicit timing information. Two sets of timing information are defined in ISO/IEC 14496-1: clock references and time stamps. The former convey the sending terminal's time base to the receiving terminal, while the latter convey a notion of relative time for specific events such as the desired decoding or composition time for portions of the encoded audio-visual information.

0.3.2 Buffer Model

The buffer model enables the sending terminal to monitor and control the buffer resources that are needed to decode each elementary stream in a presentation. The required buffer resources are conveyed to the receiving terminal by means of descriptors at the beginning of the presentation. The terminal can then decide whether or not it is capable of handling this particular presentation. The buffer model allows the sending terminal to specify when information may be removed from these buffers and enables it to schedule data transmission so that the appropriate buffers at the receiving terminal do not overflow or underflow.

0.4 Multiplexing of Streams: The Delivery Layer

The term delivery layer is used as a generic abstraction of any existing transport protocol stack that may be used to transmit and/or store content complying with ISO/IEC 14496. The functionality of this layer is not within the scope of ISO/IEC 14496-1, and only the interface to this layer is considered. This interface is the DMIF Application Interface (DAI) specified in ISO/IEC 14496-6. The DAI defines not only an interface for the delivery of streaming data, but also for signaling information required for session and channel set up as well as tear down. A wide variety of delivery mechanisms exist below this interface, with some of them indicated in Figure 1. These mechanisms serve for transmission as well as storage of streaming data, i.e., a file is considered to be a particular instance of a delivery layer. For applications where the desired transport facility does not fully address the needs of a service according to the specifications in ISO/IEC 14496, a simple multiplexing tool (M4Mux) with low delay and low overhead is defined in ISO/IEC 14496-1.

0.5 Synchronization of Streams: The Sync Layer

Elementary streams are the basic abstraction for any streaming data source. Elementary streams are conveyed as sync layer-packetized (SL-packetized) streams at the DMIF Application Interface. This packetized representation additionally provides timing and synchronization information, as well as fragmentation and random access information. The sync layer (SL) extracts this timing information to enable synchronized decoding and, subsequently, composition of the elementary stream data.

0.6 The Compression Layer

The compression layer receives data in its encoded format and performs the necessary operations to decode this data. The decoded information is then used by the terminal's composition, rendering and presentation subsystems.

0.6.1 Object Description Framework

The purpose of the object description framework is to identify and describe elementary streams and to associate them appropriately to an audio-visual scene description. Object descriptors serve to gain access to ISO/IEC 14496 content. Object content information and the interface to intellectual property management and protection systems are also part of this framework.

An object descriptor is a collection of one or more elementary stream descriptors that provide the configuration and other information for the streams that relate to either an audio-visual object, or text or font data, or a scene description. Object descriptors are themselves conveyed in elementary streams. Each object descriptor is assigned an identifier (object descriptor ID), which is unique within a defined name scope. This identifier is used to associate audio-visual objects in the scene description with a particular object descriptor, and thus the elementary streams related to that particular object.

Elementary stream descriptors include information about the source of the stream data, in form of a unique numeric identifier (the elementary stream ID) or a URL pointing to a remote source for the stream. Elementary stream descriptors also include information about the encoding format, configuration information for the decoding process and the sync layer packetization, as well as quality of service requirements for the transmission of the stream and intellectual property identification. Dependencies between streams can also be signaled within the elementary stream descriptors. This functionality may be used, for example, in scalable audio or visual object representations to indicate the logical dependency of a stream containing enhancement information, to a stream containing the base information. It can also be used to describe alternative representations for the same content (e.g. the same speech content in various languages).

0.6.1.1 Intellectual Property Management and Protection

The intellectual property management and protection (IPMP) framework for ISO/IEC 14496 content consists of a normative interface that permits an ISO/IEC 14496 terminal to host one or more IPMP Systems in the form of monolithic IPMP Systems or modular IPMP Tools. The IPMP interface consists of IPMP elementary streams and IPMP descriptors. IPMP descriptors are carried as part of an object descriptor stream. IPMP elementary streams carry time variant IPMP information that can be associated to multiple object descriptors.

The IPMP System, or IPMP Tools themselves are non-normative components that provides intellectual property management and protection functions for the terminal. The IPMP Systems or Tools uses the information carried by the IPMP elementary streams and descriptors to make protected ISO/IEC 14496 content available to the terminal.

The intellectual property management and protection (IPMP) framework for ISO/IEC 14496 content consists of a set of tools that permits an ISO/IEC 14496 terminal to support IPMP functionality. This functionality is provided by the following two different complementary technologies, supporting different levels of interoperability.

a) The IPMP framework as defined in 7.2.3, consists of a normative interface that permits an ISO/IEC 14496 terminal to host one or more IPMP Systems. The IPMP interface consists of IPMP elementary streams and IPMP descriptors. IPMP descriptors are carried as part of an object descriptor stream. IPMP elementary streams carry time variant IPMP information that can be associated to multiple object descriptors. The IPMP System itself is a non-normative component that provides intellectual property management and protection functions for the terminal. The IPMP System uses the information carried by the IPMP elementary streams and descriptors to make protected ISO/IEC 14496 content available to the terminal.

b) The IPMP framework extension, as specified in ISO/IEC 14496-13 allows, in addition to the functionality specified in ISO/IEC 14496-1, a finer granularity of governance. ISO/IEC 14496-13 provides normative support for individual IPMP components, referred to as IPMP Tools, to be normatively placed at identified points of control within the terminal systems model. Additionally ISO/IEC 14496-13 provides normative support for secure communications to be performed between IPMP Tools. ISO/IEC 14496-1 also specifies specific normative extensions at the Systems level to support the IPMP functionality described in ISO/IEC 14496-13.

An application may choose not to use an IPMP System, thereby offering no management and protection features.

0.6.1.2 Object Content Information

Object content information (OCI) descriptors convey descriptive information about audio-visual objects. The main content descriptors are: content classification descriptors, keyword descriptors, rating descriptors, language descriptors, textual descriptors, and descriptors about the creation of the content. OCI descriptors can be included directly in the related object descriptor or elementary stream descriptor or, if it is time variant, it may be carried in an elementary stream by itself. An OCI stream is organized in a sequence of small, synchronized entities called events that contain a set of OCI descriptors. OCI streams can be associated to multiple object descriptors.

0.6.2 Scene Description Streams

Scene description addresses the organization of audio-visual objects in a scene, in terms of both spatial and temporal attributes. This information allows the composition and rendering of individual audio-visual objects after the respective decoders have reconstructed the streaming data for them. For visual data, ISO/IEC 14496-11 does not mandate particular composition algorithms. Hence, visual composition is implementation dependent. For audio data, the composition process is defined in a normative manner in ISO/IEC 14496-11 and ISO/IEC 14496-3.

The scene description is represented using a parametric approach (BIFS - Binary Format for Scenes). The description consists of an encoded hierarchy (tree) of nodes with attributes and other information (including event sources and targets). Leaf nodes in this tree correspond to elementary audio-visual data, whereas intermediate nodes group this material to form audio-visual objects, and perform grouping, transformation, and other such operations on audio-visual objects (scene description nodes). The scene description can evolve over time by using scene description updates.

In order to facilitate active user involvement with the presented audio-visual information, ISO/IEC 14496-11 provides support for user and object interactions. Interactivity mechanisms are integrated with the scene description information, in the form of linked event sources and targets (routes) as well as sensors (special nodes that can trigger events based on specific conditions). These event sources and targets are part of scene description nodes, and thus allow close coupling of dynamic and interactive behavior with the specific scene at hand. ISO/IEC 14496-11, however, does not specify a particular user interface or a mechanism that maps user actions (e.g., keyboard key presses or mouse movements) to such events.

Such an interactive environment may not need an upstream channel, but ISO/IEC 14496 also provides means for client-server interactive sessions with the ability to set up upstream elementary streams and associate them to specific downstream elementary streams.

0.6.3 Audio-visual Streams

The coded representation of audio and visual information are described in ISO/IEC 14496-3 (Audio) and ISO/IEC 14496-2 (Visual) and ISO/IEC 14496-10 (Advanced Video Coding) respectively. The reconstructed audio-visual data are made available to the composition process for potential use during the scene rendering.

0.6.4 Upchannel Streams

Downchannel elementary streams may require upchannel information to be transmitted from the receiving terminal to the sending terminal (e.g., to allow for client-server interactivity). Figure 1 indicates the flowpath for an elementary stream from the receiving terminal to the sending terminal. The content of upchannel streams is specified in the same part of the specification that defines the content of the downstream data. For example, upchannel control streams for video downchannel elementary streams are defined in ISO/IEC 14496-2.

0.6.5 Interaction Streams

The coded representation of user interaction information is not in the scope of ISO/IEC 14496. But this information shall be translated into scene modification and the modifications made available to the composition process for potential use during the scene rendering.

0.6.6 Text and Font data Streams

Scene description often contains information presented in textual format. The audio-visual data encoded in the scene may also be accompanied by supplemental text information such as subtitles. In order to enable time-based updates of text data and to insure the text appearance and layout, both elementary streams carrying timed text information and font data are used. The coded representation of the timed text stream is described in ISO/IEC 14496-17. The font data format and encoded representation of font data stream are described in ISO/IEC 14496-18 (font data stream) and ISO/IEC 14496-22 (font data format).

0.7 Application Engine

The MPEG-J is a programmatic system (as opposed to a conventional parametric system) which specifies API(s) for interoperation of MPEG-4 media players with Java code. By combining MPEG-4 media and safe executable code, content creators may embed complex control and data processing mechanisms with their media data to intelligently manage the operation of the audio-visual session. The parametric MPEG-4 System forms the Presentation Engine while the MPEG-J subsystem controlling the Presentation Engine forms the Application Engine.

The Java application is delivered as a separate elementary stream to the MPEG-4 terminal. There it will be directed to the MPEG-J run time environment, from where the MPEG-J program will have access to the various components and required data of the MPEG-4 player to control it.

In addition to the basic packages of the language (java.lang, java.io, java.util) a few categories of APIs have been defined for different scopes. For the Scene graph API the objective is to provide access to the scene graph specified in ISO/IEC 14496-11: to inspect the graph, to alter nodes and their fields, and to add and remove nodes within the graph. The Resource API is used for regulation of performance: it provides a centralized facility for managing resources. This is used when the program execution is contingent upon the terminal configuration and its capabilities, both static (that do not change during execution) and dynamic. Decoder API allows the control of the decoders that are present in the terminal. The Net API provides a way to interact with the network, being compliant to the MPEG-4 DMIF Application Interface. Complex applications and enhanced interactivity are possible with these basic packages. The architecture of MPEG-J is presented in more detail in ISO/IEC 14496-11.

0.8 Extensible MPEG-4 Textual Format (XMT)

The Extensible MPEG-4 Textual (XMT) format is a textual representation of the multimedia content described in ISO/IEC 14496 using the Extensible Markup Language (XML). XMT is designed to facilitate the creation and maintenance of MPEG-4 multimedia content, whether by human authors or by automated machine programs. XMT is specified in ISO/IEC 14496-11.

The textual representation of MPEG-4 content has high-level abstractions, XMT-O, that allow authors to exchange their content easily with other authors or authoring tools, while at the same time preserving semantic intent. XMT also has low-level textual representations, XMT-A, covering the full scope and function of MPEG-4. The high-level XMT-O is designed to facilitate interoperability with the Synchronized Multimedia Integration Language (SMIL) 2.0, a recommendation from the W3C consortium, and also with Extensible 3D specification, X3D, developed by the Web3D consortium as the next generation of Virtual Reality Modeling Language (VRML).

The XMT language has grammars that are specified using the W3C XML Schema language. The grammars contain rules for element placement and attribute values, etc. These rules for XMT, defined using the Schema language, follow the binary coding rules defined in ISO/IEC 14496-11 and help ensure that the textual representation can be coded into correct binary according to ISO/IEC 14496-11 coding rules.

All constructs in the ISO/IEC 14496 specification have their parallel in the XMT textual format. For the Visual and Audio parts, XMT provides a means to reference external media streams of either pre-encoded or raw audiovisual binary content. While XMT does not contain a textual format for audiovisual media, it does contain hints in a textual format that allow an XMT tool to encode and embed the audiovisual media into a complete MPEG-4 presentation.

0.9 Patent Rights

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured the ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with the ISO and IEC. Information may be obtained from the companies listed in Annex J.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified in Annex J. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Coding of audio-visual objects —

Part 1: **Systems**

1 Scope

This part of ISO/IEC 14496 specifies system level functionalities for the communication of interactive audio-visual scenes, i.e. the coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content).

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 639-2:1998, *Codes for the representation of names of languages — Part 2: Alpha-3 code*

ISO/IEC 10646-1:2000, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

ISO/IEC 11172-2:1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 2: Video*

ISO/IEC 11172-3:1993, *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio*

ISO/IEC 13818-3:1998, *Information technology — Generic coding of moving pictures and associated audio information — Part 3: Audio*

ISO/IEC 13818-7:2006, *Information technology — Generic coding of moving pictures and associated audio information — Part 7: Advanced Audio Coding (AAC)*

ISO/IEC 14496-2:2004, *Information technology — Coding of audio-visual objects — Part 2: Visual*

ISO/IEC 14496-10:2009, *Information technology — Coding of audio-visual objects — Part 10: Advanced Video Coding*

ISO/IEC 14496-15:2004, *Information technology — Coding of audio-visual objects — Part 15: Advanced Video Coding (AVC) file format*

ISO/IEC 14496-16:2006, *Information technology — Coding of audio-visual objects — Part 16: Animation Framework eXtension (AFX)*

ISO/IEC 14496-18:2004, *Information technology — Coding of audio-visual objects — Part 18: Font compression and streaming*

ISO/IEC 13818-2:2000, *Information technology — Generic coding of moving pictures and associated audio information — Part 2: Video*

ISO/IEC 10918-1:1994, *Information technology — Digital compression and coding of continuous-tone still images — Part 1: Requirements and guidelines*

ANSI/SMPTE 291M:1996, *Television — Ancillary Data Packet and Space Formatting*

SMPTE 315M:1999, *Television — Camera Positioning Information Conveyed by Ancillary Data Packets*

W3C Recommendation: 28 October 2004 — *XML Schema*, http://www.w3.org/TR/xmlschema-0/

3 Additional references

For additional references see the Bibliography.

4 Terms and definitions

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

4.1
access unit
AU
smallest individually accessible portion of data within an **elementary stream** to which unique timing information can be attributed

4.2
alpha map
representation of the transparency parameters associated with a texture map

4.3
audio-visual object
representation of a natural or synthetic object that has an audio and/or visual manifestation

NOTE The representation corresponds to a node or a group of nodes in the BIFS scene description. Each audio-visual object is associated with zero or more **elementary streams** using one or more **object descriptors**.

4.4
audio-visual scene
AV scene
set of audio-visual objects together with scene description information that defines their spatial and temporal attributes including behaviors resulting from object and user interactions

4.5
AVC parameter set
sequence parameter set or a picture parameter set

4.6
AVC access unit
access unit made up of NAL Units as defined in ISO/IEC 14496-10 with the structure defined in ISO/IEC 14496-15:2004, 5.2.3

4.7
AVC parameter set access unit
access unit made up only of sequence parameter set NAL units or picture parameter set NAL units having same timestamps to be applied

4.8
AVC parameter set elementary stream
elementary stream containing made up only of AVC parameter set access units

4.9
AVC video elementary stream
elementary stream containing access units made up of NAL units for coded picture data

4.10
binary format for scene
BIFS
coded representation of a parametric scene description format as specified in ISO/IEC 14496-11

4.11
buffer model
model that defines how a terminal complying with ISO/IEC 14496 manages the buffer resources that are needed to decode a presentation

4.12
byte aligned
position in a coded bit stream with a distance of a multiple of 8-bits from the first bit in the stream

4.13
clock reference
special time stamp that conveys a reading of a time base

4.14
composition
process of applying scene description information in order to identify the spatio-temporal attributes and hierarchies of audio-visual objects

4.15
composition memory
CM
random access memory that contains composition units

4.16
composition time stamp
CTS
indication of the nominal composition time of a composition unit

4.17
composition unit
CU
individually accessible portion of the output that a decoder produces from access units

4.18
compression layer
layer of a system according to the specifications in ISO/IEC 14496 that translates between the coded representation of an elementary stream and its decoded representation. It incorporates the decoders

4.19
control point
point on a given elementary stream in a terminal where IPMP Processing on stream data is carried out

4.20
decoder
entity that translates between the coded representation of an elementary stream and its decoded representation

4.21
decoding buffer
DB
buffer at the input of a decoder that contains access units

4.22
decoder configuration
configuration of a decoder for processing its elementary stream data by using information contained in its elementary stream descriptor

4.23
decoding time stamp
DTS
indication of the nominal decoding time of an access unit

4.24
delivery layer
generic abstraction for delivery mechanisms (computer networks, etc.) able to store or transmit a number of multiplexed elementary streams or M4Mux streams

4.25
descriptor
data structure that is used to describe particular aspects of an elementary stream or a coded audio-visual object

4.26
DMIF application interface
DAI
interface specified in ISO/IEC 14496-6 used to model the exchange of SL-packetized stream data and associated control information between the sync layer and the delivery layer

4.27
elementary stream
ES
consecutive flow of mono-media data from a single source entity to a single destination entity on the compression layer

4.28
elementary stream descriptor
structure contained in object descriptors that describes the encoding format, initialization information, sync layer configuration, and other descriptive information about the content carried in an elementary stream

4.29
elementary stream interface
ESI
conceptual interface modeling the exchange of elementary stream data and associated control information between the compression layer and the sync layer

4.30
M4Mux channel
FMC
label to differentiate between data belonging to different constituent streams within one M4Mux stream

NOTE A sequence of data in one M4Mux channel within a M4Mux stream corresponds to one single SL-packetized stream.

4.31
M4Mux packet
smallest data entity managed by the M4Mux tool consisting of a header and a payload

4.32
M4Mux stream
sequence of M4Mux Packets with data from one or more SL-packetized streams that are each identified by their own M4Mux channel

4.33
M4Mux tool
tool that allows the interleaving of data from multiple data streams

4.34
graphics profile
profile that specifies the permissible set of graphical elements of the BIFS tool that may be used in a scene description stream

NOTE BIFS comprises both graphical and scene description elements.

4.35
inter
mode for coding parameters that uses previously coded parameters to construct a prediction

4.36
interaction stream
elementary stream that conveys user interaction information

4.37
intra
mode for coding parameters that does not make reference to previously coded parameters to perform the encoding

4.38
initial object descriptor
special object descriptor that allows the receiving terminal to gain initial access to portions of content encoded according to ISO/IEC 14496 and that conveys profile and level information to describe the complexity of the content

4.39
intellectual property identification
IPI
unique identification of one or more elementary streams corresponding to parts of one or more audio-visual objects

4.40
intellectual property management and protection system
IPMP system
generic term for mechanisms and tools to manage and protect intellectual property

NOTE This part of ISO/IEC 14496 defines the interface to such systems as well as the following.

— The provision for the identification of IPMP tools either through the use of a registration authority or through the use of a functional description of the IPMP tools' capabilities in a parametric fashion.

— Controlling the time of instantiation of IPMP tools either by the inclusion of references to the required IPMP tools or at the request of already instantiated IPMP tools.

— Providing secure messaging between IPMP tools and the terminal and between IPMP tools and the user.

— Notification of the instantiation of IPMP tools to IPMP tools requesting such notification.

— Interaction between IPMP tools, and/or the terminal and the user.

— The carriage of IPMP tools within the bitstream.

4.41
IPMP information
Information directed to a given IPMP Tool to enable, assist or facilitate its operation

4.42
IPMP system
monolithic IPMP protection scheme which requires implementation dependant access to protected streams at required Control Points and must provide any intra-communication within an IPMP System on an implementation basis

NOTE In this standard the use of the term "IPMP System" is used in some cases to indicate either an actual IPMP System or a combination of IPMP Tools whose combination provides the functionality of an IPMP System. In cases where the distinction is important the proper respective terms are used.

4.43
IPMP tool
module that performs (one or more) IPMP functions such as authentication, decryption, watermarking

NOTE Conceptually the use of one or more IPMP tools is combined to perform the functionality of an IPMP system. IPMP tools, as opposed to IPMP systems, are normatively identified as to which control points they function at as well as are provided normative methods for secure communications both within as well as outside of a given IPMP tools comprised functional "IPMP system". An additional difference between IPMP tools and IPMP systems is that IPMP tools, or a combination thereof, may be used for the protection of object streams.

4.44
IPMP tool identifier
unambiguous identifier for IPMP tools at the presentation level or at a universal level

NOTE Two different identifiers are provided to support the differentiation between the use of IPMP systems and IPMP tools.

4.45
IPMP tool list
list of selectable IPMP tools required to process the content

4.46
media node
time dependent BIFS node that refers to a media stream through a URL field in

— AnimationStream,

— AudioBuffer,

— AudioClip,

— AudioSource,

— Inline, and

— MovieTexture

4.47
media stream
one or more elementary streams whose ES descriptors are aggregated in one object descriptor and that are jointly decoded to form a representation of an AV object

4.48
media time line
time line expressing normal play back time of a media stream

4.49
MP4 file
name of the file format described in ISO/IEC 14496-14

4.50
object clock reference
OCR
clock reference that is used by a decoder to recover the time base of the encoder of an elementary stream

4.51
object content information
OCI
additional information about content conveyed through one or more elementary streams; either aggregated to individual elementary stream descriptors or is itself conveyed as an elementary stream.

4.52
object descriptor
OD
descriptor that aggregates one or more elementary streams by means of their elementary stream descriptors and defines their logical dependencies

4.53
object descriptor command
command that identifies the action to be taken on a list of object descriptors or object descriptor IDs, e.g., update or remove

4.54
object descriptor profile
profile that specifies the configurations of the object descriptor tool and the sync layer tool that are allowed

4.55
object descriptor stream
elementary stream that conveys object descriptors encapsulated in object descriptor commands

4.56
object time base
OTB
time base valid for a given elementary stream, and hence for its decoder; conveyed to the decoder via object clock references and which is used by all time stamps relating to this object's decoding process

4.57
parametric audio decoder
set of tools for representing and decoding speech signals coded at bit rates between 6 Kbps and 16 Kbps, according to the specifications in ISO/IEC 14496-3

4.58
parametric description
SDL declaration that describes the parametric configuration and other interface message(s) that drive the tool and the behaviour defined for fulfilment of such a description

4.59
quality of service
QoS
performance that an elementary stream requests from the delivery channel through which it is transported. QoS is characterized by a set of parameters (bit rate, delay jitter, bit error rate, etc.)

4.60
random access
process of beginning to read and decode a coded representation at an arbitrary point within the elementary stream

4.61
reference point
location in the data or control flow of a system that has some defined characteristics

4.62
rendering
action of transforming a scene description and its constituent audio-visual objects from a common representation space to a specific presentation device (i.e. speakers and a viewing window)

4.63
rendering area
portion of the display device's screen into which the scene description and its constituent audio-visual objects are to be rendered

4.64
scene description
information that describes the spatio-temporal positioning of audio-visual objects as well as their behavior resulting from object and user interactions and which makes reference to elementary streams with audio-visual data by means of pointers to object descriptors

4.65
scene description stream
elementary stream that conveys scene description information

4.66
scene graph elements
elements of the BIFS language that relate only to the structure of the audio-visual scene (spatio-temporal positioning of audio-visual objects as well as their behavior resulting from object and user interactions) excluding the audio, visual and graphics nodes as specified in ISO/IEC 14496-11

4.67
scene graph profile
profile that defines the permissible set of scene graph elements of the BIFS tool that may be used in a scene description stream

NOTE BIFS comprises both graphical and scene description elements.

4.68
seekable
property of a media stream for which the play back is possible from any position

4.69
SL-packetized stream
SPS
sequence of sync layer packets that encapsulate one elementary stream

4.70
stream object
media stream or a segment thereof, referenced through a URL field in the scene in the form “OD:n” or “OD:n#<segmentName>”

4.71
structured audio
method of describing synthetic sound effects and music as defined by ISO/IEC 14496-3

4.72
sync layer
SL
layer to adapt elementary stream data for communication across the DMIF Application Interface, providing timing and synchronization information, as well as fragmentation and random access information

NOTE The sync layer syntax is configurable and can be configured to be empty.

4.73
sync layer configuration
configuration of the sync layer syntax for a particular elementary stream using information contained in its elementary stream descriptor

4.74
sync layer packet
SL-packet
smallest data entity managed by the sync layer consisting of a configurable header and a payload which may consist of one complete access unit or a partial access unit

4.75
syntactic description language SDL
language defined in ISO/IEC 14496-1:2010, Clause 8 that allows the description of a bitstream's syntax

4.76
systems decoder model
SDM
model that provides an abstract view of the behavior of a terminal compliant to ISO/IEC 14496 which consists of the buffer model and the timing model

4.77
system time base
STB
time base of the terminal whose resolution is implementation-dependent and according to which all operations in the terminal are performed

4.78
terminal
system that sends, or receives and presents the coded representation of an interactive audio-visual scene as defined by ISO/IEC 14496-11 which can be a standalone system, or part of an application system complying with ISO/IEC 14496

4.79
time base
clock, equivalent to a counter that is periodically incremented

4.80
timing model
model that specifies the semantic meaning of timing information, how it is incorporated (explicitly or implicitly) in the coded representation of information, and how it can be recovered at the receiving terminal

4.81
time stamp
indication of a particular time instant relative to a time base

4.82
track
collection of related samples in an MP4 file

5 Abbreviated terms

AU	access unit
AV	audio-visual
AVC	advanced video coding (see ISO/IEC 14496-10)
BIFS	binary format for scene
CM	composition memory
CTS	composition time stamp
CU	composition unit
DAI	DMIF application interface (see ISO/IEC 14496-6)
DB	decoding buffer
DTS	decoding time stamp
ES	elementary stream
ESI	elementary stream interface
ESID	elementary stream identifier
FMC	M4Mux channel
IP	intellectual property
IPI	intellectual property identification
IPMP	intellectual property management and protection
NAL	network abstraction layer
OCI	object content information
OCR	object clock reference
OD	object descriptor
ODID	object descriptor identifier
OTB	object time base
PLL	phase locked loop
QOS	quality of service
SDL	syntactic description language
SDM	systems decoder model
SEI	supplementary enhancement information
SL	synchronization layer
SL-packet	synchronization layer packet
SPS	SL-packetized stream
STB	system time base
URL	universal resource locator
VOP	video object plane

6 Conventions

For the purpose of unambiguously defining the syntax of the various bitstream components defined by the normative parts of ISO/IEC 14496 a *syntactic description language* is used. This language allows the specification of the mapping of the various parameters in a binary format as well as how they are placed in a serialized bitstream. The definition of the language is provided in Clause 8 of this specification.

7 Streaming Framework

7.1 Systems Decoder Model

7.1.1 Introduction

The purpose of the systems decoder model (SDM) is to provide an abstract view of the behavior of a terminal complying with ISO/IEC 14496. It may be used by the sender to predict how the receiving terminal will behave in terms of buffer management and synchronization when decoding data received in the form of elementary streams. The systems decoder model includes a timing model and a buffer model.

The systems decoder model specifies:

1. the interface for accessing demultiplexed data streams (DMIF Application Interface),
2. decoding buffers for coded data for each elementary stream,
3. the behavior of elementary stream decoders,
4. composition memory for decoded data from each decoder, and
5. the output behavior of composition memory towards the compositor.

These elements are depicted in Figure 2. Each elementary stream is attached to one single decoding buffer. More than one elementary stream may be connected to a single decoder (e.g., in a decoder of a scalable audio-visual object).

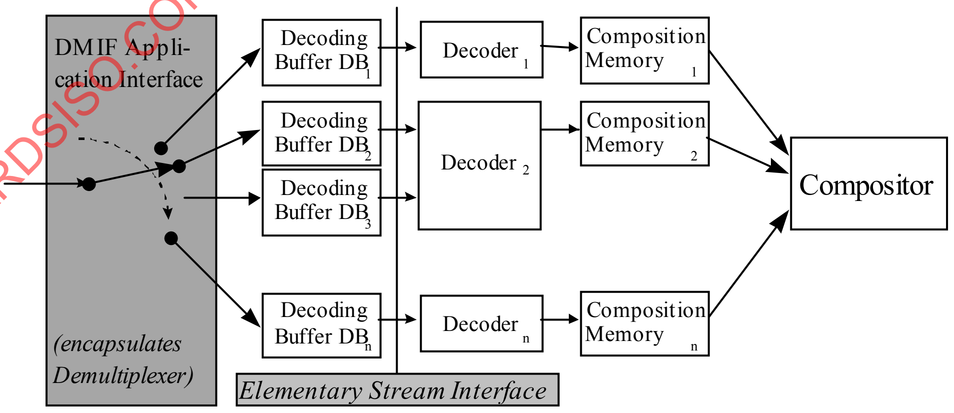


Figure 2 — Systems Decoder Model

7.1.2 Concepts of the systems decoder model

This Subclause defines the concepts necessary for the specification of the timing and buffering model. The sequence of definitions corresponds to a walk from the left to the right side of the SDM illustration in Figure 2.

7.1.2.1 DMIF Application Interface (DAI)

For the purposes of the systems decoder model, the DMIF Application Interface encapsulates the demultiplexer and provides access to streaming data that is consumed by the decoding buffers. The streaming data received through the DAI consists of SL-packetized streams. The required properties of the DAI are described in 7.3.3. The DAI semantics are fully specified in ISO/IEC 14496-6.

7.1.2.2 SL-Packetized Stream (SPS)

An SL-packetized stream consists of a sequence of packets, according to the syntax and semantics specified in 7.3.2, that encapsulate a single elementary stream. The packets contain elementary stream data partitioned in access units as well as side information, e.g., for timing and access unit labeling. SPS data payload enters the decoding buffers, i.e., the side information is removed at the input to the decoding buffers.

7.1.2.3 Access Units (AU)

Elementary stream data is partitioned into access units. The delineation of an access unit is completely determined by the entity that generates the elementary stream (e.g., the compression layer). An access unit is the smallest data entity to which timing information can be attributed. Two access units from the same elementary stream shall never refer to the same decoding or composition time. Any further partitioning of the data in an elementary stream is not visible for the purposes of the systems decoder model. Access units are conveyed by SL-packetized streams and are received by the decoding buffers. The decoders consume access units with the necessary side information (e.g., time stamps) from the decoding buffers.

NOTE — An ISO/IEC 14496-1 compliant terminal implementation is not required to process each incoming access unit as a whole. It is furthermore possible to split an access unit into several fragments for transmission as specified in 7.3. This allows the sending terminal to dispatch partial AUs immediately as they are generated during the encoding process. Such partial AUs may have significance for improved error resilience.

7.1.2.4 Decoding Buffer (DB)

The decoding buffer is a buffer at the input of an elementary stream decoder in the receiving terminal that receives and stores access units. The systems buffer model enables the sending terminal to monitor the decoding buffer resources that are used during a presentation.

7.1.2.5 Elementary Streams (ES)

Streaming data received at the output of a decoding buffer, independent of its content, is considered as an elementary stream for the purpose of ISO/IEC 14496. The elementary streams are produced and consumed by the compression layer entities (encoders and decoders, respectively). ISO/IEC 14496 assumes that the integrity of an elementary stream is preserved from end to end.

7.1.2.6 Elementary Stream Interface (ESI)

The elementary stream interface is a concept that models the exchange of elementary stream data and associated control information between the compression layer and the sync layer. It is explained further in 7.3.

7.1.2.7 Decoder

For the purposes of this model, the decoder extracts access units from the decoding buffer at precisely defined points in time and places composition units, the results of the decoding processes, in the composition memory. A decoder may be attached to several decoding buffers.

7.1.2.8 Composition Units (CU)

Decoders consume access units and produce composition units. An access unit corresponds to an integer number of composition units. In case of multiple elementary streams attached to a single decoder (scalable coding), each composition unit is derived from access units from one or more of these streams. Composition units reside in composition memory.

7.1.2.9 Composition Memory (CM)

The composition memory is a random access memory that contains composition units. The size of this memory is not normatively specified.

7.1.2.10 Compositor

The compositor takes composition units out of the composition memory and either consumes them (e.g. composes and presents them, in the case of audio-visual data) or skips them. The compositor is not specified in ISO/IEC 14496-1, as the details of this operation are not relevant within the context of the systems decoder model. 7.1.3.5 defines which composition units are available to the compositor at any instant of time.

7.1.3 Timing Model Specification

The timing model relies on clock references and time stamps to synchronize audio-visual data conveyed by one or more elementary streams. The concept of a clock with its associated clock references is used to convey the notion of time to a receiving terminal. Time stamps are used to indicate the precise time instants at which the receiving terminal consumes the access units in the decoding buffers or may access the composition units resident in the composition memory. The time stamps are therefore associated with access units and composition units. The semantics of the timing model are defined in the subsequent clauses. The syntax for conveying timing information is specified in 7.3.2.

NOTE — This timing model is designed for rate-controlled (“push”) applications.

7.1.3.1 System Time Base (STB)

The system time base (STB) defines the terminal’s notion of time. The resolution of the STB is implementation dependent. All actions of the terminal are scheduled according to this time base for the purpose of this timing model.

NOTE — This does not imply that all terminals compliant with ISO/IEC 14496 operate on one single STB.

7.1.3.2 Object Time Base (OTB)

The object time base (OTB) defines the notion of time for a given data stream. The resolution of this OTB can be selected as required by the application or as defined by a profile. All time stamps that the sending terminal inserts in a coded data stream refer to this time base. The OTB of a data stream is known at the receiving terminal either by means of object clock reference information inserted in the stream or by an indication that its time base is slaved to a time base conveyed with another stream, as specified in 7.3.2.3.

NOTE 1 — Elementary streams may be created for the sole purpose of conveying time base information.

NOTE 2 — The receiving terminal’s system time base need not be locked to any of the available object time bases.

7.1.3.3 Object Clock Reference (OCR)

A special kind of time stamps, object clock references (OCR), are used to convey the OTB to the elementary stream decoder. The value of the OCR corresponds to the value of the OTB at the time the sending terminal generates the object clock reference time stamp. OCR time stamps are placed in the SL packet header as described in 7.3.2.4. The receiving terminal shall evaluate the OCR when its last bit is extracted at the input of the decoding buffer.

7.1.3.4 Decoding Time Stamp (DTS)

Each access unit has an associated nominal decoding time, the time at which it must be available in the decoding buffer for decoding. The AU is not guaranteed to be available in the decoding buffer either before or after this time. Decoding is assumed to occur instantaneously when the instant of time indicated by the DTS is reached.

This point in time can be implicitly specified if the (constant) temporal distance between successive access units is indicated in the setup of the elementary stream (see 7.3.2.3). Otherwise a decoding time stamp (DTS) whose syntax is defined in 7.3.2.4 conveys this point in time.

A decoding time stamp shall only be conveyed for an access unit that carries a composition time stamp as well, and only if the DTS and CTS values are different. Presence of both time stamps in an AU may indicate a reversal between coding order and composition order.

7.1.3.5 Composition Time Stamp (CTS)

Each composition unit has an associated nominal composition time, the time at which it must be available in the composition memory for composition. The CU is not guaranteed to be available in the composition memory *for composition* before this time. Since the SDM assumes an instantaneous decoding process, the CU is available to the *decoder*, at that instant in time corresponding to the DTS of the corresponding AU, for further use (e.g. in prediction processes).

This instant in time is implicitly known, if the (constant) temporal distance between successive composition units is indicated in the setup of the elementary stream. Otherwise a composition time stamp (CTS) whose syntax is defined in 7.3.2.4 conveys this instant in time.

The current CU is instantaneously accessible by the compositor anytime between its composition time and the composition time of the subsequent CU. If a subsequent CU does not exist, the current CU becomes unavailable at the end of the lifetime of its elementary stream (i.e., when its elementary stream descriptor is removed).

In case of audio decoders, the following additionally applies to the audio samples within a composition unit: the composition time applies to the *n*-th audio sample within the composition unit. The value of *n* is 1 unless explicitly specified in ISO/IEC 14496-3, 1.6.6 *Interface between Audio and Systems*.

7.1.3.6 Occurrence and Precision of Timing Information in Elementary Streams

The frequency at which DTS, CTS and OCR values are to be inserted in the bitstream as well as the precision, jitter and drift are application and profile dependent. Some usage considerations can be found in 7.3.2.7.

7.1.3.7 Time Stamps for Dependent Elementary Streams

An audio-visual object may refer to multiple elementary streams that constitute a scalable content representation (see 7.2.7.1.5). Such a set of elementary streams shall adhere to a single object time base. Temporally co-located access units for such elementary streams are then identified by identical DTS or CTS values.

EXAMPLE

The example in Figure 3 illustrates the arrival of two access units at the Systems Decoder. Due to the constant delay assumption of the model (see 7.1.4.2 below), the arrival times correspond to the instants in time when the sending terminal has sent the respective AUs. The sending terminal must select this instant in time so that the Decoding Buffer at the receiving terminal never overflows or underflows. At the receiving terminal, an AU is instantaneously decoded, at that instant in time corresponding to its DTS, and the resulting CU(s) are placed in the composition memory and remain there until the subsequent CU(s) arrive or the associated object descriptor is removed.

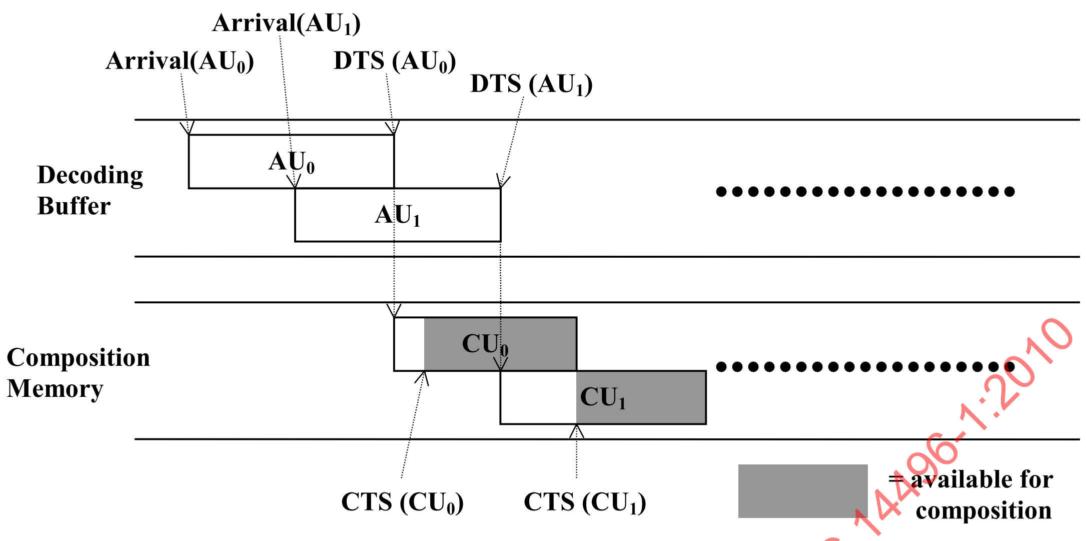
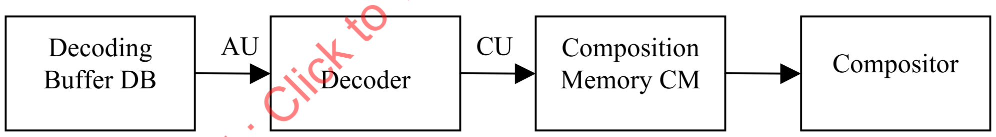


Figure 3 — Composition unit availability

7.1.4 Buffer Model Specification

7.1.4.1 Elementary Decoder Model

Figure 4 indicates one branch of the systems decoder model (Figure 2). This simplified model is used to specify the buffer model. It treats each elementary stream separately and therefore, associates a composition memory with only one decoder. The legend following Figure 4 elaborates on the symbols used in this figure.



Legend:

DB	Decoding buffer for the elementary stream.
CM	Composition memory for the elementary stream.
AU	The current access unit input to the decoder.
CU	The current composition unit input to the composition memory. CU results from decoding AU. There may be several composition units resulting from decoding one access unit.

Figure 4 — Flow diagram for the systems decoder model

7.1.4.2 Assumptions

7.1.4.2.1 Constant end-to-end delay

Data transmitted in real time have a timing model in which the end-to-end delay from the encoder input at the sending terminal, to the decoder output at the receiving terminal, is constant. This delay is equal to the sum of the delay due to the encoding process, subsequent buffering, multiplexing at the sending terminal, the delay due to the delivery layers and the delay due to the demultiplexing, decoder buffering and decoding processes at the receiving terminal.

Note that the receiving terminal is free to add a temporal offset (delay) to the absolute values of all time stamps if it can cope with the additional buffering needed. However, the temporal difference between two time stamps (that determines the temporal distance between the associated AUs or CUs) has to be preserved for real-time performance.

NOTE — Two elementary streams that adhere to different time bases may be synchronized tightly in case of constant end-to-end delay as assumed by this model. If an application cannot implement this model assumption, such tight synchronization may not be achievable. Tolerances for the constant end-to-end delay assumption need to be defined through the profile and level mechanism.

7.1.4.2.2 Demultiplexer

The end-to-end delay between multiplexer output, at the sending terminal, and demultiplexer input, at the receiving terminal, is constant.

7.1.4.2.3 Decoding Buffer

The needed decoding buffer size is known by the sending terminal and conveyed to the receiving terminal as specified in 7.2.6.6.

The size of the decoding buffer is measured in bytes.

The decoding buffer is filled at the rate given by the maximum bit rate for this elementary stream while data is available and with a zero rate otherwise. The maximum bit rate is conveyed by the sending terminal as a part of the decoder configuration information during the set up phase for each elementary stream (see 7.2.6.6).

Information is received from the DAI in the form of SL packets. The SL packet headers are removed at the input to the decoding buffers.

7.1.4.2.4 Decoder

The decoding processes are assumed to be instantaneous for the purposes of the systems decoder model.

7.1.4.2.5 Composition Memory

The mapping of an AU to one or more CUs (by the decoder) is known implicitly at both the sending and the receiving terminals.

7.1.4.2.6 Compositor

The composition processes are assumed to be instantaneous for the purposes of the systems decoder model.

7.1.4.3 Managing Buffers: A Walkthrough

In this example, we assume that the model is used in a “push” scenario. In applications where non-real time content is to be delivered, flow control by suitable signaling may be established to request access units at the time they are needed at the receiving terminal. The mechanisms for doing so are application-dependent, and are not specified in ISO/IEC 14496.

The behaviors of the various elements in the SDM are modeled as follows:

- The sending terminal signals the required decoding buffer resources to the receiving terminal before starting the delivery. This is done as specified in 7.2.6.6 either explicitly, by requesting the decoding buffer sizes for individual elementary streams, or implicitly, by indicating a profile (see Clause 9). The decoding buffer size is measured in bytes.

- The sending terminal models the behavior of the decoding buffers by making the following assumptions :
- Each decoding buffer is filled at the maximum bitrate specified for its associated elementary stream as long as data is available.
- At the instant of time corresponding to its DTS, an AU is instantaneously decoded and removed from the decoding buffer.
- At the instant of time corresponding to its DTS, a known amount of CUs corresponding to the just decoded AU are put in the composition memory.

The current CU is available to the compositor between instants of time corresponding to the CTS of the current CU and the CTS of the subsequent CU. If a subsequent CU does not exist, the current CU becomes unavailable at the end of lifetime of its data stream.

Using these assumptions on the buffer model, the sending terminal may freely use the space in the decoding buffers. For example, it may deliver data for several AUs of a stream, for non real time usage, to the receiving terminal, and pre-store them in the DB long before they have to be decoded (assuming sufficient space is available). Subsequently, the full delivery bandwidth may be used to transfer data of a real time stream just in time. The composition memory may be used, for example, as a reordering buffer. In the case of visual decoding, it may contain the decoded P-frames needed by a video decoder for the decoding of intermediate B-frames, before the arrival of the CTS of the latest P-frame.

7.2 Object Description Framework

7.2.1 Introduction

The scene description (specified in ISO/IEC 14496-11) and the elementary streams that convey streaming data are the basic building blocks of the architecture of ISO/IEC 14496-1. Elementary streams carry data for audio or visual objects as well as for the scene description itself. The object description framework provides the link between elementary streams and the scene description. The scene description declares the spatio-temporal relationship of audio-visual objects, while the object description framework specifies the elementary stream resources that provide the time-varying data for the scene. This indirection facilitates independent changes to the scene structure, the properties of the elementary streams (e.g. its encoding) and their delivery.

The object description framework consists of a set of descriptors that allows to identify, describe and properly associate elementary streams to each other and to audio-visual objects used in the scene description. Numeric identifiers, called ObjectDescriptorIDs, associate object descriptors to appropriate nodes in the scene description. Object descriptors are themselves conveyed in elementary streams to allow time stamped changes to the available set of object descriptors to be made.

Each object descriptor is itself a collection of descriptors that describe one or more elementary streams that are associated to a single node and that usually relate to a single audio or visual object. This allows to indicate a scalable content representation as well as multiple alternative streams that convey the same content, e.g., in multiple qualities or different languages.

An elementary stream descriptor within an object descriptor identifies a single elementary stream with a numeric identifier, called ES_ID. Each elementary stream descriptor contains the information necessary to initiate and configure the decoding process for the elementary stream, as well as intellectual property identification. Optionally, additional information may be associated to a single elementary stream, most notably quality of service requirements for its transmission or a language indication. Both, object descriptors and elementary stream descriptors may use URLs to point to remote object descriptors or a remote elementary stream source, respectively.

The object description framework provides the hooks to implement intellectual property management and protection (IPMP) systems. IPMP information is conveyed both through IPMP descriptors as part of the object descriptor stream and through IPMP streams that carry time variant IPMP information. The structure of IPMP descriptors and IPMP streams is specified in this Clause while their internal syntax and semantics and, hence, the operation of the IPMP system is outside the scope of ISO/IEC 14496.

Object content information allows the association of metadata with a whole presentation or with individual object descriptors or with elementary stream descriptors. A set of OCI descriptors is defined that either form an integral part of an object descriptor or elementary stream descriptor or are conveyed by means of a proper OCI stream that allows the conveyance of time variant object content information.

Access to ISO/IEC 14496 content is gained through an initial object descriptor that needs to be made available through means not defined in ISO/IEC 14496. The initial object descriptor in the simplest case points to the scene description stream and the corresponding object descriptor stream. The access scenario is outlined in 7.2.7.3.

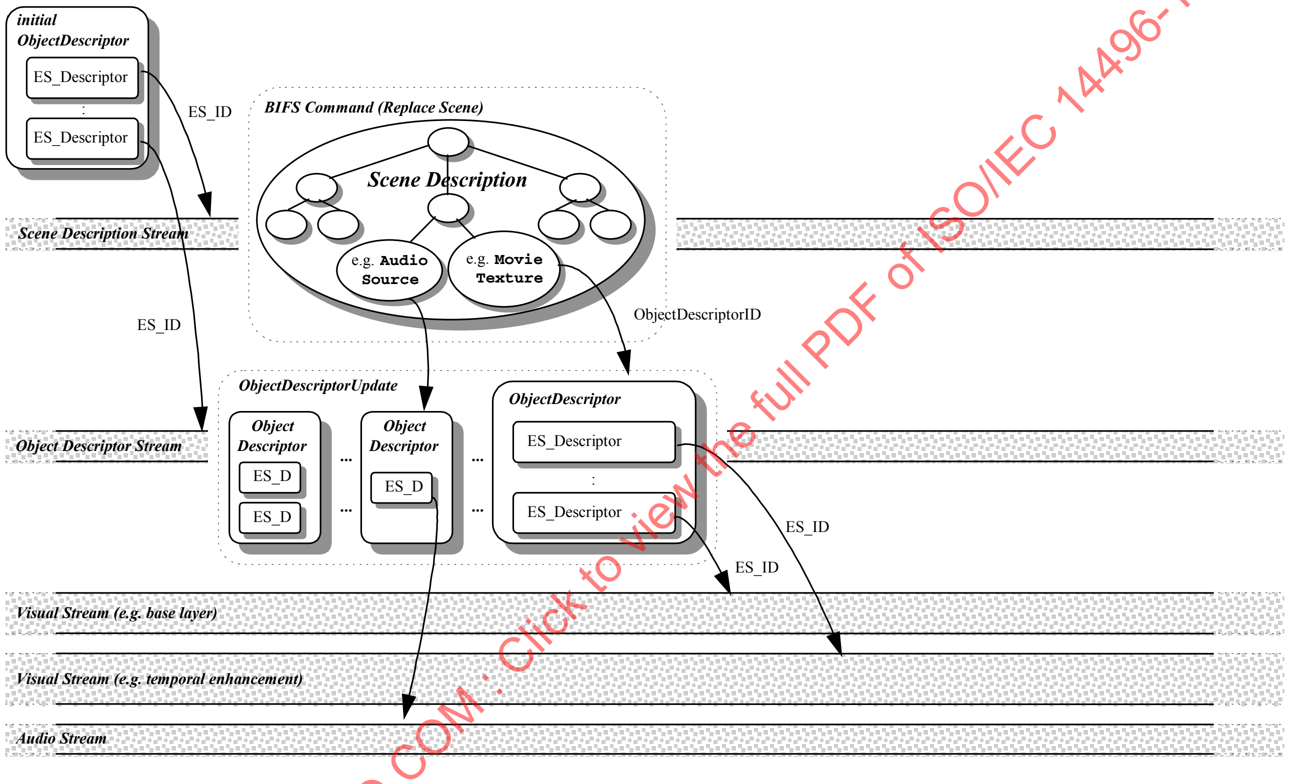


Figure 5 — Object descriptors linking scene description to elementary streams

The remainder of this Clause is structured in the following way:

- 7.2.2 specifies the data structures on which the object descriptor framework is based.
- 7.2.3 specifies the concepts of the IPMP elements in the object description framework.
- 7.2.4 specifies the object content information elements in the object description framework.
- 7.2.5 specifies the object descriptor stream and the syntax and semantics of the command set that allows the update or removal of object descriptor components.
- 7.2.6 specifies the syntax and semantics of the object descriptor and its component descriptors.
- 7.2.7 specifies rules for object descriptor usage as well as the procedure to access content through object descriptors.
- 7.2.8 specifies the usage of the IPMP system interface.

7.2.2 Common data structures

7.2.2.1 Overview

The commands and descriptors defined in this Subclause constitute self-describing classes, identified by unique class tags. Each class encodes explicitly its size in bytes. This facilitates future compatible extensions of the commands and descriptors. A class may be expanded with additional syntax elements that are ignored by an OD decoder that expects an earlier revision of a class. In addition, anywhere in a syntax where a set of tagged classes is expected it is permissible to intersperse expandable classes with unknown class tag values. These classes shall be skipped, using the encoded size information.

The remainder of this Clause defines the syntax and semantics of the command and descriptor classes. Some commands and descriptors contain themselves a set of component descriptors. They are said to *aggregate a set of component descriptors.*

Table 1 — List of Class Tags for Descriptors

Tag value	Tag name
0x00	Forbidden
0x01	ObjectDescrTag
0x02	InitialObjectDescrTag
0x03	ES_DescrTag
0x04	DecoderConfigDescrTag
0x05	DecSpecificInfoTag
0x06	SLConfigDescrTag
0x07	ContentIdentDescrTag
0x08	SupplContentIdentDescrTag
0x09	IPI_DescrPointerTag
0x0A	IPMP_DescrPointerTag
0x0B	IPMP_DescrTag
0x0C	QoS_DescrTag
0x0D	RegistrationDescrTag
0x0E	ES_ID_IncTag
0x0F	ES_ID_RefTag
0x10	MP4_IOD_Tag
0x11	MP4_OD_Tag
0x12	IPL_DescrPointerRefTag
0x13	ExtensionProfileLevelDescrTag
0x14	profileLevelIndicationIndexDescrTag
0x15-0x3F	Reserved for ISO use
0x40	ContentClassificationDescrTag
0x41	KeyWordDescrTag
0x42	RatingDescrTag
0x43	LanguageDescrTag
0x44	ShortTextualDescrTag
0x45	ExpandedTextualDescrTag
0x46	ContentCreatorNameDescrTag
0x47	ContentCreationDateDescrTag
0x48	OCICreatorNameDescrTag
0x49	OCICreationDateDescrTag
0x4A	SmpteCameraPositionDescrTag
0x4B	SegmentDescrTag

Tag value	Tag name
0x4C	MediaTimeDescrTag
0x4D-0x5F	Reserved for ISO use (OCI extensions)
0x60	IPMP_ToolsListDescrTag
0x61	IPMP_ToolTag
0x62	M4MuxTimingDescrTag
0x63	M4MuxCodeTableDescrTag
0x64	ExtSLConfigDescrTag
0x65	M4MuxBufferSizeDescrTag
0x66	M4MuxIdentDescrTag
0x67	DependencyPointerTag
0x68	DependencyMarkerTag
0x69	M4MuxChannelDescrTag
0x6A-0xBF	Reserved for ISO use
0xC0-0xFE	User private
0xFF	Forbidden

7.2.2.2 BaseDescriptor

7.2.2.2.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class BaseDescriptor : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

7.2.2.2.2 Semantics

This class is an abstract base class that is extended by the descriptor classes specified in 7.2.6. Each descriptor constitutes a self-describing class, identified by a unique class tag. This abstract base class establishes a common name space for the class tags of these descriptors. The values of the class tags are defined in Table 1. As an expandable class the size of each class instance in bytes is encoded and accessible through the instance variable sizeOfInstance (see 8.3.3).

A class that allows the aggregation of classes of type BaseDescriptor may actually aggregate any of the classes that extend BaseDescriptor.

NOTE — User private descriptors may have an internal structure, for example to identify the country or manufacturer that uses a specific descriptor. The tags and semantics for such user private descriptors may be managed by a registration authority if required.

The following additional symbolic names are introduced:

ExtDescrTagStartRange = 0x6A

ExtDescrTagEndRange = 0xFE

OCIDescrTagStartRange = 0x40

OCIDescrTagEndRange = 0x5F

7.2.2.3 BaseCommand

7.2.2.3.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class BaseCommand : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

7.2.2.3.2 Semantics

This class is an abstract base class that is extended by the command classes specified in 7.2.5.5. Each command constitutes a self-describing class, identified by a unique class tag. This abstract base class establishes a common name space for the class tags of these commands. The values of the class tags are defined in Table 2. As an expandable class the size of each class instance in bytes is encoded and accessible through the instance variable sizeOfInstance (see 8.3.3).

Table 2 — List of Class Tags for Commands

Tag value	Tag name
0x00	forbidden
0x01	ObjectDescrUpdateTag
0x02	ObjectDescrRemoveTag
0x03	ES_DescrUpdateTag
0x04	ES_DescrRemoveTag
0x05	IPMP_DescrUpdateTag
0x06	IPMP_DescrRemoveTag
0x07	ES_DescrRemoveRefTag
0x08	ObjectDescrExecuteTag
0x09-0xBF	Reserved for ISO (command tags)
0xC0-0xFE	User private
0xFF	forbidden

A class that allows the aggregation of classes of type BaseCommand may actually aggregate any of the classes that extend BaseCommand.

NOTE — User private commands may have an internal structure, for example to identify the country or manufacturer that uses a specific command. The tags and semantics for such user private command may be managed by a registration authority if required.

7.2.3 Intellectual Property Management and Protection Framework (IPMP)

7.2.3.1 Overview

The intellectual property management and protection (IPMP) framework for ISO/IEC 14496 content consists of a normative interface that permits an ISO/IEC 14496 terminal to host one or more IPMP Systems or IPMP Tools. Additionally, the framework contains a secure messaging system usable between IPMP Tools as well as IPMP Tools and the Terminal and IPMP Tools and the User which is specified in ISO/IEC 14496-13.

An IPMP System or IPMP Tools are non-normative components that provide intellectual property management and protection functions for the terminal.

The IPMP interface consists of IPMP elementary streams and IPMP descriptors. The normative structure of IPMP elementary streams is specified in this Subclause. IPMP descriptors are carried as part of an object descriptor stream and are specified in 7.2.6.14. The IPMP interface allows applications (or derivative application standards) to build specialized IPMP Systems or IPMP Tools. Alternatively, an application may choose not to use an IPMP System or IPMP Tools, thereby offering no management and protection features. The IPMP System and IPMP Tools use the information carried by the IPMP elementary streams and descriptors to make protected ISO/IEC 14496 content available to the terminal. The detailed semantics and decoding process of the IPMP System or IPMP Tools are not in the scope of ISO/IEC 14496. The usage of the IPMP System/Tools Interface, however, is explained in 7.2.8 with the usage of the IPMP framework being explained.

7.2.3.2 IPMP Streams

7.2.3.2.1 Structure of the IPMP Stream

The IPMP stream is an elementary stream that passes time-varying information to one or more IPMP Systems or Tools. This is accomplished by periodically sending a sequence of IPMP messages along with the content at a period determined by the IPMP System(s) or Tool(s).

7.2.3.2.2 Access Unit Definition

An IPMP access unit consists of one or more IPMP messages, as defined in 7.2.3.2.5. All IPMP messages that are to be processed at the same instant in time shall constitute a single access unit. Access units in IPMP streams shall be labeled and time-stamped by suitable means. This shall be done via the related flags and the composition time stamps, respectively, in the SL packet header (see 7.3.2.4). The composition time indicates the point in time at which an IPMP access unit becomes valid, i.e., when the embedded IPMP messages shall be evaluated. Decoding and composition time for an IPMP access unit shall always have the same value.

An access unit does not necessarily convey or update the complete set of IPMP messages that are currently required. In that case it just modifies the persistent state of the IPMP system. However, if an access unit conveys the complete set of IPMP messages required at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to ‘1’ for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to ‘0’.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no IPMP messages in it indicates that at the current time instant no IPMP messages are required for operation.

7.2.3.2.3 Time Base for IPMP Streams

The time base associated to an IPMP stream shall be indicated by suitable means. This shall be done by means of object clock reference time stamps in the SL packet headers (see 7.3.2.4) for this stream or by indicating the elementary stream from which this IPMP stream inherits the time base (see 7.3.2.3). All time stamps in the SL-packetized IPMP stream refer to this time base.

An IPMP stream shall adhere to the same time base as the one or more content elementary streams to which it is associated (see 7.2.8). Consequently, an IPMP stream may not be associated to multiple content elementary streams that themselves adhere to different time bases.

7.2.3.2.4 IPMP Decoder Configuration

7.2.3.2.4.1 Syntax

```
class IPMPDecoderConfiguration extends DecoderSpecificInfo : bit(8)
tag=DecSpecificInfoTag {
   // IPMP system specific configuration information
}
```

7.2.3.2.4.2 Semantics

An IPMP system may require information to initialize its operation. This information shall be conveyed by extending the `decoderSpecificInfo` class as specified in 7.2.6.7. If utilized, `IPMPDecoderConfiguration` shall be conveyed in the `ES_Descriptor` declaring the IPMP stream.

7.2.3.2.5 IPMP message syntax and semantics

7.2.3.2.5.1 Syntax

```
aligned(8) expandable(2^28-1) class IPMP_Message
{
  bit(16) IPMPS_Type;
  if (IPMPS_Type == 0)
  (
    bit(8) URLString[sizeOfInstance-2];
  )
  else (if (IPMPS_Type == 0xFFFF)
  (
    bit(16) IPMP_DescriptorIDEx;
    IPMP_Data_BaseClass IPMP_ExtendedData[]
  } else {
    bit(8) IPMP_data[sizeOfInstance-2];
  }
}
```

7.2.3.2.5.2 Semantics

The `IPMP_Message` conveys time-varying IPMP information for associated IPMP System or IPMP Tool instances.

`IPMPS_Type` – The type of the IPMP System, in “Hooks” compliant Terminals as specified in ISO/IEC 14496-1. The values “0x0002” to “0x2000” are reserved for future ISO use. A Registration Authority, as designated by ISO/IEC JTC 1, shall assign a unique valid value for this field for a specific IPMP System Type. If the `IPMP_DescriptorID` is “0”, another URL is referenced. This process continues until an `IPMP_Message` with a non-zero `IPMP_DescriptorID` is accessed.

`URLString[]` - contains a UTF-8 [6] encoded URL that shall point to the location of a remote `IPMP_Message`.

`IPMP_DescriptorID` – this is one of the `IPMP_DescriptorIDs` in the scope of service of this IPMP Stream and identifies the recipient(s) of the `IPMP_Message`.

`IPMP_ExtendedData` - The IPMP data that is extended from `IPMP_Data_BaseClass` to be delivered to the IPMP tool.

`IPMP_data` - opaque data to be delivered to the IPMP Tool.

The `IPMP_Message` is backward compatible with the `IPMP_Message` of ISO/IEC 14496-1:2001. However, in order to unambiguously identify the version of the IPMP stream, the `ObjectTypeIndication` shall be set to “0x02” for streams complying with this part of the specification. IPMP Streams complying with ISO/IEC 14496-1 shall use an `ObjectTypeIndication` of “0xFF” as specified for in 7.2.6.6.2.

7.2.3.2.6 Extension tags for the IPMP_Data_BaseClass

7.2.3.2.6.1 IPMP_Data_BaseClass

The `IPMP_Data_BaseClass` is intended to be extended to provide the carriage of ISO defined as well as user defined IPMP related data.

7.2.3.2.6.2 Syntax

```
abstract aligned(8) expandable(2^28-1) class IPMP_Data_BaseClass:
  bit(8) tag=0…255
{
  bit(8) Version;
  bit(32)dataID;
  // Fields and data extending this message.
}
```

7.2.3.2.6.3 Semantics

`Version` - indicates the version of syntax used in the IPMP Data and shall be set to “0x01”.

`dataID` – used for the purpose of identifying the message. Tools replying directly to a message shall include the same `dataID` in any response.

`tag` indicates the tag for the extended IPMP data. The exact values for the extension tags are defined in ISO/IEC 14496-13.

IPMP data extending from `IPMP_Data_BaseClass` can be carried in the following three places:

- `IPMP_Descriptor`
- `IPMP_Message` defined in ISO/IEC 14496-13 which is subsequently carried in IPMP Stream.
- Messages defined in ISO/IEC 14496-13 specified to carry messages between IPMP tools.

7.2.4 Object Content Information (OCI)

7.2.4.1 Overview

Audio-visual objects that are associated with elementary stream data through an object descriptor may have additional object content information attached to them. For this purpose, a set of OCI descriptors is defined in 7.2.6.18. OCI descriptors may directly be included as part of an object descriptor or `ES_Descriptor` as defined in 7.2.6.

In order to accommodate time variant OCI that is separable from the object descriptor stream, OCI descriptors may as well be conveyed in an OCI stream. An OCI stream is referred to through an ES_Descriptor, with the `streamType` field set to OCI_Stream. How OCI streams may be aggregated to object descriptors is defined in 7.2.7.1.3. The structure of the OCI stream is defined in this Subclause.

7.2.4.2 OCI Streams

7.2.4.2.1 Structure of the OCI Stream

The OCI stream is an elementary stream that conveys time-varying object content information, termed OCI events. Each OCI event consists of a number of OCI descriptors.

7.2.4.2.2 Access Unit Definition

An OCI access unit consists of one or more OCI_Events, as described in 7.2.4.2.5. Access units in OCI elementary streams shall be labelled and time stamped by suitable means. This shall be done by means of the related flags and the composition time stamp, respectively, in the SL packet header (see 7.3.2.4). The composition time indicates the point in time when an OCI access unit becomes valid, i.e., when the embedded OCI events shall be added to the list of events. Decoding and composition time for an OCI access unit shall always have the same value.

An access unit may or may not convey or update the complete set of OCI events that are currently valid. In the latter case, it just modifies the persistent state of the OCI decoder. However, if an access unit conveys the complete set of OCI events valid at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to '1' for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to '0'.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no OCI events in it indicates that at the current time instant no valid OCI events exist.

7.2.4.2.3 Time Base for OCI Streams

The time base associated with an OCI stream shall be indicated by suitable means. This shall be done by the use of object clock reference time stamps in the SL packet headers (see 7.3.2.4) for this stream or by indicating the elementary stream from which this OCI stream inherits the time base (see 7.3.2.3). All time stamps in the SL-packetized OCI stream refer to this time base.

7.2.4.2.4 OCI Decoder Configuration

7.2.4.2.4.1 Syntax

```
class OCIDecoderConfiguration extends DecoderSpecificInfo  : bit(8)
tag=DecSpecificInfoTag {
   const bit(8) versionLabel = 0x01;
}
```

7.2.4.2.4.2 Semantics

This information is needed to initialize operation of the OCI decoder. It shall be conveyed by extending the `decoderSpecificInfo` class as specified in 7.2.6.7. `OCIDecoderConfiguration` shall be conveyed in the `ES_Descriptor` declaring the OCI stream.

`versionLabel` – indicates the version of OCI specification used on the corresponding OCI data stream. Only the value 0x01 is allowed; all the other values are reserved.

7.2.4.2.5 OCI_Events syntax and semantics

7.2.4.2.5.1 Syntax

```
aligned(8) expandable(2^28-1) class OCI_Event {
   bit(15) eventID;
   bit(1)  absoluteTimeFlag;
   bit(32) startingTime;
   bit(32) duration;
   OCI_Descriptor OCI_Descr[1 .. 255];
}
```

7.2.4.2.5.2 Semantics

`eventID` – contains the identification number of the described event that is unique within the scope of this OCI stream.

`absoluteTimeFlag` – indicates the time base for `startingTime` as described below.

`startingTime` – indicates the starting time of the event in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.

EXAMPLE — 02:36:45:89 is coded as "0x023645" concatenated with "0b0101.1001" (89 in binary), resulting to "0x02364559".

If `absoluteTimeFlag` is set to zero, `startingTime` is relative to the object time base of the corresponding object. In that case it is the responsibility of the application to ensure that this object time base is conveyed such that `startingTime` can be identified unambiguously (see 7.3.2.7). If `absoluteTimeFlag` is set to one, `startingTime` is expressed as an absolute value, refering to wall clock time.

`duration` – contains the duration of the corresponding object in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.

`OCI_Descr[]` – an array of one up to 255 `OCI_Descriptor` classes as specified in 7.2.6.18.2.

7.2.5 Object Descriptor Stream

7.2.5.1 Structure of the Object Descriptor Stream

Similar to the scene description, object descriptors are transported in a dedicated elementary stream, termed object descriptor stream. Within such a stream, it is possible to dynamically convey, update and remove complete object descriptors, or their component descriptors, the ES_Descriptors, and IPMP descriptors. The update mechanism allows, for example, to advertise new elementary streams for an audio-visual object as they become available, or to remove references to streams that are no longer available. Updates are time stamped to indicate the instant in time they take effect.

This Subclause specifies the structure of the object descriptor elementary stream including the syntax and semantics of its constituent elements, the object descriptor commands (OD commands).

7.2.5.2 Access Unit Definition

An OD access unit consists of one or more OD commands, as described in 7.2.5.5. All OD commands that are to be processed at the same instant in time shall constitute a single access unit. Access units in object descriptor elementary streams shall be labelled and time stamped by suitable means. This shall be done by means of the related flags and the composition time stamp, respectively, in the SL packet header (see 7.3.2.4). The composition time indicates the point in time when an OD access unit becomes valid, i.e., when the embedded OD commands shall be executed. Decoding and composition time for an OD access unit shall always have the same value.

An access unit may not convey or update the complete set of object descriptors that are currently required. In that case it just modifies the persistent state of the object descriptor decoder. However, if an access unit conveys the complete set of object descriptors required at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to '1' for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to '0'.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no OD commands in it indicates that at the current time instant no valid object descriptors exist.

7.2.5.3 Time Base for Object Descriptor Streams

The time base associated to an object descriptor stream shall be indicated by suitable means. This shall be done by means of object clock reference time stamps in the SL packet headers (see 7.3.2.4) for this stream or by indicating the elementary stream from which this object descriptor stream inherits the time base (see 7.3.2.3). All time stamps in the SL-packetized object descriptor stream refer to this time base.

7.2.5.4 OD Decoder Configuration

The object descriptor decoder does not require additional configuration information.

7.2.5.5 OD Command Syntax and Semantics

7.2.5.5.1 Overview

Object descriptors and their components as defined in 7.2.6 shall always be conveyed as part of one of the OD commands specified in this Subclause. The commands describe the action to be taken on the components conveyed with the command, specifically 'update' or 'remove'. Each command affects one or more object descriptors, ES_Descriptors or IPMP descriptors.

7.2.5.5.2 ObjectDescriptorUpdate

7.2.5.5.2.1 Syntax

```
class ObjectDescriptorUpdate extends BaseCommand : bit(8)
tag=ObjectDescrUpdateTag {
  ObjectDescriptorBase OD[0 .. 255];
}
```

7.2.5.5.2.2 Semantics

The `ObjectDescriptorUpdate` class conveys a list of new or updated object descriptors. If an object descriptor is updated, the streams refered to by the old object descriptor shall be closed and the streams refered to by the new object descriptor may be accessed by the content access procedure (see 7.2.7.3.6.2).

NOTE - The ES_DescriptorUpdate or ES_DescriptorRemove commands may be used to add or remove individual ES_Descriptors of an existing object descriptor.

`OD[]` – an array of object descriptors as defined in 7.2.6.3 and 7.2.6.4. The array shall have any number of one up to 255 elements.

7.2.5.5.3 ObjectDescriptorRemove

7.2.5.5.3.1 Syntax

```
class ObjectDescriptorRemove extends BaseCommand : bit(8)
tag=ObjectDescrRemoveTag {
  bit(10) objectDescriptorId[(sizeOfInstance*8)/10];
}
```

7.2.5.5.3.2 Semantics

The `ObjectDescriptorRemove` class renders unavailable a set of object descriptors. The BIFS nodes associated to these object descriptors shall have no reference any more to the elementary streams that have been listed in the removed object descriptors. An objectDescriptorID that does not refer to a valid object descriptor is ignored.

NOTE — It is possible that a scene description node references an OD_ID which does not currently have an associated OD.

ObjectDescriptorId[] – an array of ObjectDescriptorIDs that indicates the object descriptors that are removed.

7.2.5.5.4 ES_DescriptorUpdate

7.2.5.5.4.1 Syntax

```
class ES_DescriptorUpdate extends BaseCommand : bit(8) tag=ES_DescrUpdateTag {
   bit(10) objectDescriptorId;
   ES_Descriptor esDescr[1 .. 255];
}
```

7.2.5.5.4.2 Semantics

The ES_DescriptorUpdate class conveys a list of new ES_Descriptors for the object descriptor labeled objectDescriptorID. ES_Descriptors with ES_IDs that have already been received within the same name scope shall be ignored.

To update the characterstics of an elementary stream, it is required that its original ES_Descriptor be removed and the changed ES_Descriptor be conveyed.

When an IPMP stream is added, the affected elementary streams, as defined in 7.2.8.2, shall be processed under the new IPMP conditions starting at the point in time that this ES_DescriptorUpdate command becomes valid (see 7.2.5.2).

ES_DescriptorUpdate shall not be applied on object descriptors that have set URL_Flag to '1' (see 7.2.6.3).

An elementary stream identified with a given ES_ID may be attached to more than one object descriptor. All corresponding ES_Descriptors refering to this ES_ID that are conveyed through either ES_DescriptorUpdate or ObjectDescriptorUpdate commands shall have identical content.

objectDescriptorID - identifies the object descriptor for which ES_Descriptors are updated. If the objectDescriptorID does not refer to any valid object descriptor, then this command is ignored.

esDescr[] – an array of ES_Descriptors as defined in 7.2.6.5. The array shall have any number of one up to 255 elements.

7.2.5.5.5 ES_DescriptorRemove

7.2.5.5.5.1 Syntax

```
class ES_DescriptorRemove extends BaseCommand : bit(8) tag=ES_DescrRemoveTag {
   bit(10) objectDescriptorId;
   aligned (8) bit(16) ES_ID[1..255];
}
```

7.2.5.5.5.2 Semantics

The ES_DescriptorRemove class removes the reference to an elementary stream from an object descriptor and renders this stream unavailable for nodes referencing this object descriptor.

When an IPMP stream is removed, the affected elementary streams, as defined in 7.2.8.2, shall be processed under the new IPMP conditions starting at the point in time that this ES_DescriptorRemove command becomes valid (see 7.2.5.2).

ES_DescriptorRemove shall not be applied on object descriptors that have set URL_Flag to '1' (see 7.2.6.3).

objectDescriptorID - identifies the object descriptor from which ES_Descriptors are removed. If the objectDescriptorID does not refer to a valid object descriptor in the same scope, then this command is ignored.

ES_ID[] – an array of ES_IDs that labels the ES_Descriptors to be removed from objectDescriptorID. If any of the ES_IDs do not refer to an ES_Descriptor currently referenced by the OD, then those ES_IDs are ignored. The array shall have any number of one up to 255 elements.

7.2.5.5.6 IPMP_DescriptorUpdate

7.2.5.5.6.1 Syntax

```
class IPMP_DescriptorUpdate extends BaseCommand : bit(8) tag=IPMP_DescrUpdateTag
{
   IPMP_Descriptor  ipmpDescr[1..255];
}
```

7.2.5.5.6.2 Semantics

The IPMP_DescriptorUpdate class conveys a list of new or updated IPMP_Descriptors. An IPMP_Descriptor identified by an IPMP_DescriptorID that has already been received within the same name scope shall be replaced by the new descriptor.

Updates to an IPMP_Descriptor shall be propagated at the time this IPMP_DescriptorUpdate becomes valid (see 7.2.5.2) to all IPMP Systems that refer to this IPMP_Descriptor through an IPMP_DescriptorPointer (see 7.2.6.13). The handling of the descriptors by the IPMP systems is not normative.

IPMP_Descriptors remain valid until they are replaced by another IPMP_DescriptorUpdate command or removed.

ipmpDescr[] – an array of IPMP_Descriptor as specified in 7.2.6.14.

7.2.5.5.7 IPMP_DescriptorRemove

7.2.5.5.7.1 Syntax

```
class IPMP_DescriptorRemove extends BaseCommand : bit(8) tag=IPMP_DescrRemoveTag
{
   bit(8) IPMP_DescriptorID[1..255];
}
```

7.2.5.5.7.2 Semantics

The `IPMP_DescriptorRemove` class conveys a list of `IPMP_DescriptorsIDs` that identify the `IPMP_Descriptors` that shall be removed.

The removal of IPMP_Descriptors shall be notified to all IPMP systems at the time this IPMP_DescriptorRemove becomes valid (see 7.2.5.2). The handling of the descriptors by the IPMP systems is not normative.

`IPMP_DescriptorID[]` – is a list of `IPMP_DescriptorIDs`.

7.2.5.5.8 ObjectDescriptorExecute

7.2.5.5.8.1 Syntax

```
class ObjectDescriptorExecute extends BaseCommand : bit(8) tag=
ObjectDescriptorExecuteTag {
     bit(10) objectDescriptorId[(sizeOfInstance*8)/10];
}
```

7.2.5.5.8.2 Semantics

The `ObjectDescriptorExecute` class instructs the terminal that Elementary streams contained therein shall be opened as the server will transmit data on one or more of the streams. Failure by the terminal to comply may result in data loss and/or other undefined behavior.

7.2.6 Object Descriptor Components

7.2.6.1 Overview

Object descriptors contain various additional descriptors as their components, in order to describe individual elementary streams and their properties. They shall always be conveyed as part of one of the OD commands specified in the previous Subclause. This Subclause defines the syntax and semantics of object descriptors and their component descriptors.

7.2.6.2 ObjectDescriptorBase

7.2.6.2.1 Syntax

```
abstract class ObjectDescriptorBase extends BaseDescriptor : bit(8)
tag=[ObjectDescrTag..InitialObjectDescrTag] {
// empty. To be filled by classes extending this class.
}
```

7.2.6.2.2 Semantics

This is an abstract base class for the different types of object descriptor classes defined subsequently. The term “object descriptor” is used to generically refer to any such derived object descriptor class or instance thereof.

7.2.6.3 ObjectDescriptor

7.2.6.3.1 Syntax

```
class ObjectDescriptor extends ObjectDescriptorBase : bit(8) tag=ObjectDescrTag {
  bit(10) ObjectDescriptorID;
  bit(1) URL_Flag;
  const bit(5) reserved=0b1111.1;
  if (URL_Flag) {
    bit(8) URLlength;
    bit(8) URLstring[URLlength];
  } else {
    ES_Descriptor esDescr[1 .. 255];
    OCI_Descriptor ociDescr[0 .. 255];
    IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
    IPMP_Descriptor ipmpDescr [0 .. 255];
  }
  ExtensionDescriptor extDescr[0 .. 255];
}
```

When an ObjectDescriptor is used in the OD track of an MP4 file, the ObjectDescrTag is replaced by MP4_OD_Tag.

7.2.6.3.2 Semantics

The `ObjectDescriptor` consists of three different parts.

The first part uniquely labels the object descriptor within its name scope (see 7.2.7.2.4) by means of an `objectDescriptorId`. Nodes in the scene description use `objectDescriptorID` to refer to the related object descriptor. An optional `URLstring` indicates that the actual object descriptor resides at a remote location.

The second part consists of a list of `ES_Descriptors`, each providing parameters for a single elementary as well as an optional set of object content information descriptors and pointers to IPMP descriptors for the contents for elementary stream content described in this object descriptor.

The third part is a set of optional descriptors that support the inclusion of future extensions as well as the transport of private data in a backward compatible way.

`objectDescriptorId` – This syntax element uniquely identifies the `ObjectDescriptor` within its name scope. The value 0 is forbidden and the value 1023 is reserved.

`URL_Flag` – a flag that indicates the presence of a `URLstring`.

`URLlength` – the length of the subsequent `URLstring` in bytes.

`URLstring[]` – A string with a UTF-8 (ISO/IEC 10646-1) encoded URL that shall point to another `ObjectDescriptor`. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. Within the current name scope, the new object descriptor shall be referenced by the `objectDescriptorId` of the object descriptor carrying the URLstring. On name scopes see 7.2.7.2.4. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

`esDescr[]` – an array of `ES_Descriptors` as defined in 7.2.6.5. The array shall have any number of one up to 255 elements.

`ociDescr[]` – an array of `OCI_Descriptors`, as defined in 7.2.6.18.2, that relates to the audio-visual object(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

ipmpDescrPtr[] – an array of IPMP_DescriptorPointer, as defined in 7.2.6.13, that points to the IPMP_Descriptors related to the elementary stream(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

ipmpDescr[] – a list of IPMP_Descriptors that may be referenced by streams declared in esDescr. The array shall have any number of zero up to 255 elements. The following scope and usage rules apply:

i. Entries in the ipmpDescr table define IPMP System/Tools that can be referenced by IPMP_DescriptorPointers located in the OD itself or ESDs declared in this OD.

ii. OD contained IPMP_Descriptors have scope within the given OD only and shall not be referenced by subsequently declared IODs, ODs, streams nor available for updating via IPMP_DescriptorUpdates.

iii. The OD contained IPMP_Descriptors shall not be referenced by IODs, ODs or streams declared in OD declared OD or Scene streams.

extDescr[] – an array of ExtensionDescriptors as defined in 7.2.6.16. The array shall have any number of zero up to 255 elements.

7.2.6.4 InitialObjectDescriptor

7.2.6.4.1 Syntax

```
class InitialObjectDescriptor extends ObjectDescriptorBase : bit(8)
tag=InitialObjectDescrTag {
  bit(10) ObjectDescriptorID;
  bit(1) URL_Flag;
  bit(1) includeInlineProfileLevelFlag;
  const bit(4) reserved=0b1111;
  if (URL_Flag) {
    bit(8) URLlength;
    bit(8) URLstring[URLlength];
  } else {
    bit(8) ODProfileLevelIndication;
    bit(8) sceneProfileLevelIndication;
    bit(8) audioProfileLevelIndication;
    bit(8) visualProfileLevelIndication;
    bit(8) graphicsProfileLevelIndication;
    ES_Descriptor esDescr[1 .. 255];
    OCI_Descriptor ociDescr[0 .. 255];
    IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
    IPMP_Descriptor ipmpDescr [0 .. 255];
    IPMP_ToolListDescriptor toolListDescr[0 .. 1];
  }
  ExtensionDescriptor extDescr[0 .. 255];
}
```

When an InitialObjectDescriptor is used in the OD track in an MP4 file, the InitialObjectDescrTag is replaced by MP4_IOD_Tag.

7.2.6.4.2 Semantics

The InitialObjectDescriptor is a variation of the ObjectDescriptor specified in the previous Subclause that allows to signal profile and level information for the content refered by it. It shall be used to gain initial access to ISO/IEC 14496 content (see 7.2.7.3).

Profile and level information indicated in the `InitialObjectDescriptor` indicates the profile and level supported by at least the first base layer stream (i.e. an elementary stream with a `streamDependenceFlag` set to 0) in each object descriptor depending on this initial object descriptor.

`objectDescriptorId` – This syntax element uniquely identifies the `InitialObjectDescriptor` within its name scope (see 7.2.7.2.4). The value 0 is forbidden and the value 1023 is reserved.

`URL_Flag` – a flag that indicates the presence of a `URLstring`.

`includeInlineProfileLevelFlag` – a flag that, if set to one, indicates that the subsequent profile indications take into account the resources needed to process any content that might be inlined.

`URLlength` – the length of the subsequent `URLstring` in bytes.

`URLstring[]` – A string with a UTF-8 (ISO/IEC 10646-1) encoded URL that shall point to another `InitialObjectDescriptor`. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. Within the current name scope, the new object descriptor shall be referenced by the `objectDescriptorId` of the object descriptor carrying the URLstring. On name scopes see 7.2.7.2.4. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

`ODProfileLevelIndication` – an indication as defined in Table 3 of the object descriptor profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 3 — ODProfileLevelIndication Values

Value	Profile	Level
0x00	Forbidden	-
0x01	Reserved for ISO use (no SL extension)	-
0x02-0x7F	Reserved for ISO use (SL extension)	-
0x03-0x7F	Reserved for ISO use	
0x80-0xFD	user private	-
0xFE	No OD profile specified	-
0xFF	No OD capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any OD profile specified in ISO/IEC 14496-1. Usage of the value 0xFF indicates that none of the OD profile capabilities are required for this content. Usage of the value 0x01 also indicates that the SL extension mechanism is not present .		

`sceneProfileLevelIndication` – an indication as defined in ISO/IEC 14496-11 of the scene graph profile and level required to process the content associated with this `InitialObjectDescriptor`.

`audioProfileLevelIndication` – an indication as defined in ISO/IEC 14496-3 of the audio profile and level required to process the content associated with this `InitialObjectDescriptor`.

`visualProfileLevelIndication` – an indication as defined in ISO/IEC 14496-2 and in Table 4 of the visual profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 4 — visualProfileLevelIndication Values

Value	Profile	Level
0x00-0x7E	defined in ISO/IEC 14496-2 Annex G	-
0x7F	ISO/IEC 14496-10 Advanced Video Coding	-
0x80-0xFD	defined in ISO/IEC 14496-2 Annex G	-
0xFE	no visual profile specified	-
0xFF	no visual capability required	

NOTE 1 Usage of the value 0x7F indicates the use of any profile and level of ISO/IEC 14496-10 AVC. For the real profile and level numbers for ISO/IEC 14496-10 refer to the `DecoderSpecificInfo`.

NOTE 2 Usage of the value 0xFE indicates that the content described by this `InitialObjectDescriptor` does not comply to any visual profile specified in ISO/IEC 14496-2 or -10. Usage of the value 0xFF indicates that none of the visual profile capabilities are required for this content.

`graphicsProfileLevelIndication` – an indication as defined in ISO/IEC 14496-11 of the graphics profile and level required to process the content associated with this `InitialObjectDescriptor`.

`esDescr[]` – an array of `ES_Descriptors` as defined in 7.2.6.5. The array shall have any number of one up to 255 elements.

`ociDescr[]` – an array of OCI_Descriptors as defined in 7.2.6.18 that relates to the set of audio-visual objects that are described by this initial object descriptor. The array shall have any number of zero up to 255 elements.

`ipmpDescrPtr[]` – an array of `IPMP_DescriptorPointer`, as defined in 7.2.6.13, that points to the IPMP_Descriptors related to the elementary stream(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

`ipmpDescr []` – a list of `IPMP_Descriptors` that may be referenced by streams declared in `esDescr`. The array shall have any number of zero up to 255 elements. The following scope and usage rules apply:

i. Entries in the ipmpDescr table define IPMP System/Tools that can be referenced by `IPMP_DescriptorPointers` located in the IOD itself or ESDs declared in this IOD.

ii. IOD contained `IPMP_`Descriptors have scope within the given IOD only and shall not be referenced by subsequently declared IODs, ODs, streams nor available for updating via `IPMP_DescriptorUpdates`.

iii. The IOD contained `IPMP_Descriptors` shall not be referenced by IODs, ODs, streams declared in IOD declared OD or Scene streams.

`toolListDescr` – a list of all IPMP tools required for the processing of the content. The array shall have zero or 1 element.

`extDescr[]` – an array of `ExtensionDescriptors` as defined in 7.2.6.16. The array shall have any number of zero up to 255 elements.

7.2.6.5 ES_Descriptor

7.2.6.5.1 Syntax

```
class ES_Descriptor extends BaseDescriptor : bit(8) tag=ES_DescrTag {
   bit(16) ES_ID;
   bit(1) streamDependenceFlag;
   bit(1) URL_Flag;
   bit(1) OCRstreamFlag;
   bit(5) streamPriority;
   if (streamDependenceFlag)
      bit(16) dependsOn_ES_ID;
   if (URL_Flag) {
      bit(8) URLlength;
      bit(8) URLstring[URLlength];
   }
   if (OCRstreamFlag)
      bit(16) OCR_ES_Id;
   DecoderConfigDescriptor decConfigDescr;
   if (ODProfileLevelIndication==0x01)      //no SL extension.
   {
      SLConfigDescriptor slConfigDescr;
   }
   else                                    // SL extension is possible.
   {
      SLConfigDescriptor slConfigDescr;
   }
   IPI_DescrPointer ipiPtr[0 .. 1];
   IP_IdentificationDataSet ipIDS[0 .. 255];
   IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
   LanguageDescriptor langDescr[0 .. 255];
   QoS_Descriptor qosDescr[0 .. 1];
   RegistrationDescriptor regDescr[0 .. 1];
   ExtensionDescriptor extDescr[0 .. 255];
}
```

7.2.6.5.2 Semantics

The `ES_Descriptor` conveys all information related to a particular elementary stream and has three major parts.

The first part consists of the `ES_ID` which is a unique reference to the elementary stream within its name scope (see 7.2.7.2.4), a mechanism to describe dependencies of elementary streams within the scope of the parent object descriptor and an optional URL string. Dependencies and usage of URLs are specified in 7.2.7.

The second part consists of the component descriptors which convey the parameters and requirements of the elementary stream.

The third part is a set of optional extension descriptors that support the inclusion of future extensions as well as the transport of private data in a backward compatible way.

`ES_ID` – This syntax element provides a unique label for each elementary stream within its name scope. The values 0 and 0xFFFF are reserved.

`streamDependenceFlag` – If set to one indicates that a `dependsOn_ES_ID` will follow.

`URL_Flag` – if set to 1 indicates that a `URLstring` will follow.

`OCRstreamFlag` – indicates that an `OCR_ES_ID` syntax element will follow.

streamPriority – indicates a relative measure for the priority of this elementary stream. An elementary stream with a higher streamPriority is more important than one with a lower streamPriority. The absolute values of streamPriority are not normatively defined.

dependsOn_ES_ID – is the ES_ID of another elementary stream on which this elementary stream depends. The stream with dependsOn_ES_ID shall also be associated to the same object descriptor as the current ES_Descriptor.

URLlength – the length of the subsequent URLstring in bytes.

URLstring[] – contains a UTF-8 (ISO/IEC 10646-1) encoded URL that shall point to the location of an SL-packetized stream by name. The parameters of the SL-packetized stream that is retrieved from the URL are fully specified in this ES_Descriptor. See also 7.2.7.3.3. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

OCR_ES_ID – indicates the ES_ID of the elementary stream within the name scope (see 7.2.7.2.4) from which the time base for this elementary stream is derived. Circular references between elementary streams are not permitted.

decConfigDescr – is a DecoderConfigDescriptor as specified in 7.2.6.6.

slConfigDescr – is an SLConfigDescriptor as specified in 7.2.6.8. If ODProfileLevelIndication is different from 0x01, it may be an extension of SLConfigDescriptor (i.e. and extended class) as defined in 7.2.6.8.

ipiPtr[] – an array of zero or one IPI_DescrPointer as specified in 7.2.6.12.

ipIDS[] – an array of zero or more IP_IdentificationDataSet as specified in 7.2.6.9.

Each ES_Descriptor shall have either one IPI_DescrPointer or zero up to 255 IP_IdentificationDataSet elements. This allows to unambiguously associate an IP Identification to each elementary stream.

ipmpDescrPtr[] – an array of IPMP_DescriptorPointer, as defined in 7.2.6.13, that points to the IPMP_Descriptors related to the elementary stream described by this ES_Descriptor. The array shall have any number of zero up to 255 elements.

langDescr[] – an array of zero or one LanguageDescriptor structures as specified in 7.2.6.18.6. It indicates the language attributed to this elementary stream.

NOTE — Multichannel audio streams may be treated as one elementary stream with one ES_Descriptor by ISO/IEC 14496. In that case different languages present in different channels of the multichannel stream are not identifyable with a LanguageDescriptor.

qosDescr[] – an array of zero or one QoS_Descriptor as specified in 7.2.6.15.

extDescr[] – an array of ExtensionDescriptor structures as specified in 7.2.6.16.

7.2.6.6 DecoderConfigDescriptor

7.2.6.6.1 Syntax

```
class DecoderConfigDescriptor extends BaseDescriptor : bit(8)
tag=DecoderConfigDescrTag {
   bit(8) objectTypeIndication;
   bit(6) streamType;
   bit(1) upStream;
   const bit(1) reserved=1;
   bit(24) bufferSizeDB;
   bit(32) maxBitrate;
   bit(32) avgBitrate;
   DecoderSpecificInfo decSpecificInfo[0 .. 1];
   profileLevelIndicationIndexDescriptor profileLevelIndicationIndexDescr
[0..255];
}
```

7.2.6.6.2 Semantics

The `DecoderConfigDescriptor` provides information about the decoder type and the required decoder resources needed for the associated elementary stream. This is needed at the receiving terminal to determine whether it is able to decode the elementary stream. A stream type identifies the category of the stream while the optional decoder specific information descriptor contains stream specific information for the set up of the decoder in a stream specific format that is opaque to this layer.

`ObjectTypeIndication` – an indication of the object or scene description type that needs to be supported by the decoder for this elementary stream as per Table 5.

Table 5 — objectTypeIndication Values

Value	`ObjectTypeIndication` Description
0x00	Forbidden
0x01	Systems ISO/IEC 14496-1 [a]
0x02	Systems ISO/IEC 14496-1 [b]
0x03	Interaction Stream
0x04	Systems ISO/IEC 14496-1 Extended BIFS Configuration [c]
0x05	Systems ISO/IEC 14496-1 AFX [d]
0x06	Font Data Stream
0x07	Synthesized Texture Stream
0x08	Streaming Text Stream
0x09-0x1F	reserved for ISO use
0x20	Visual ISO/IEC 14496-2 [e]
0x21	Visual ITU-T Recommendation H.264 | ISO/IEC 14496-10 [f]
0x22	Parameter Sets for ITU-T Recommendation H.264 | ISO/IEC 14496-10 [f]
0x23-0x3F	reserved for ISO use
0x40	Audio ISO/IEC 14496-3 [g]
0x41-0x5F	reserved for ISO use
0x60	Visual ISO/IEC 13818-2 Simple Profile
0x61	Visual ISO/IEC 13818-2 Main Profile
0x62	Visual ISO/IEC 13818-2 SNR Profile
0x63	Visual ISO/IEC 13818-2 Spatial Profile
0x64	Visual ISO/IEC 13818-2 High Profile
0x65	Visual ISO/IEC 13818-2 422 Profile
0x66	Audio ISO/IEC 13818-7 Main Profile
0x67	Audio ISO/IEC 13818-7 LowComplexity Profile
0x68	Audio ISO/IEC 13818-7 Scaleable Sampling Rate Profile
0x69	Audio ISO/IEC 13818-3
0x6A	Visual ISO/IEC 11172-2
0x6B	Audio ISO/IEC 11172-3
0x6C	Visual ISO/IEC 10918-1
0x6D	reserved for registration authority [i]

Value	`ObjectTypeIndication` Description
0x6E	Visual ISO/IEC 15444-1
0x6F - 0x9F	reserved for ISO use
0xA0 - 0xBF	reserved for registration authority [i]
0xC0 - 0xE0	user private
0xE1	reserved for registration authority [i]
0xE2 - 0xFE	user private
0xFF	no object type specified [h]

[a] This type is used for all 14496-1 streams unless specifically indicated to the contrary. Scene Description scenes, which are identified with StreamType=0x03, using this object type value shall use the BIFSConfig specified in ISO/IEC 14496-11.

[b] This object type shall be used, with StreamType=0x03, for Scene Description streams that use the BIFSv2Config specified in ISO/IEC 14496-11. Its use with other StreamTypes is reserved.

[c] This object type shall be used, with StreamType=0x03, for Scene Description streams that use the BIFSConfigEx specified in 7.2.6.7 of this specification. Its use with other StreamTypes is reserved.

[d] This object type shall be used, with StreamType=0x03, for Scene Description streams that use the AFXConfig specified in 7.2.6.7 of this specification. Its use with other StreamTypes is reserved.

[e] Includes associated Amendment(s) and Corrigendum(a). The actual object types are defined in ISO/IEC 14496-2 and are conveyed in the DecoderSpecificInfo as specified in ISO/IEC 14496-2, Annex K.

[f] Includes associated Amendment(s) and Corrigendum(a). The actual object types are defined in ITU-T Recommendation H.264 | ISO/IEC 14496-10 and are conveyed in the DecoderSpecificInfo as specified in this amendment, I.2.

[g] Includes associated Amendment(s) and Corrigendum(a). The actual object types are defined in ISO/IEC 14496-3 and are conveyed in the DecoderSpecificInfo as specified in ISO/IEC 14496-3 subpart 1 subclause 6.2.1.

[h] Streams with this value with a StreamType indicating a systems stream (values 1,2,3, 6, 7, 8, 9) shall be treated as if the ObjectTypeIndication had been set to 0x01.

[i] The latest entries registered can be found at http://www.mp4ra.org/object.html.

When the objectTypeIndication value is 0x6C (Visual ISO/IEC 10918-1, which is JPEG) the stream may contain one or more Access Units, where one Access Unit is defined to be a complete JPEG (as defined in Visual ISO/IEC 10918-1). Note, that timing and other Access Unit and packetization information is to be carried in the transport layer such as the MPEG-4 Sync Layer.

When the objectTypeIndication value is 0x6E (Visual ISO/IEC 15444-1, which is JPEG 2000) the stream may contain one or more Access Units, where one Access Unit is defined to be a complete JPEG 2000 (as defined in Visual ISO/IEC 15444-1). Note, that timing and other Access Unit and packetization information is to be carried in the transport layer such as the MPEG-4 Sync Layer.

NOTE The format defined in ISO/IEC 15444-3 is preferred for the storage of JPEG 2000 sequences in file format of the ISO/IEC 14496-12 family, including MP4.

`streamType` – conveys the type of this elementary stream as per Table 6.

Table 6 — streamType Values

streamType value	**Stream type description**
0x00	Forbidden
0x01	ObjectDescriptorStream (see 7.2.5)
0x02	ClockReferenceStream (see 7.3.2.5)
0x03	SceneDescriptionStream (see ISO/IEC 14496-11)
0x04	VisualStream
0x05	AudioStream
0x06	MPEG7Stream
0x07	IPMPStream (see 7.2.3.2)
0x08	ObjectContentInfoStream (see 7.2.4.2)
0x09	MPEGJStream
0x0A	Interaction Stream
0x0B	IPMPToolStream (see [ISO/IEC 14496-13])
0x0C - 0x1F	reserved for ISO use
0x20 - 0x3F	user private

`upStream` – indicates that this stream is used for upstream information.

`bufferSizeDB` – is the size of the decoding buffer for this elementary stream in byte.

`maxBitrate` – is the maximum bitrate in bits per second of this elementary stream in any time window of one second duration.

`avgBitrate` – is the average bitrate in bits per second of this elementary stream. For streams with variable bitrate this value shall be set to zero.

`decSpecificInfo[]` – an array of zero or one decoder specific information classes as specified in 7.2.6.7.

`ProfileLevelIndicationIndexDescr [0..255]` – an array of unique identifiers for a set of profile and level indications as carried in the `ExtensionProfileLevelDescr` defined in 7.2.6.19.

7.2.6.7 DecoderSpecificInfo

7.2.6.7.1 Syntax

```
abstract class DecoderSpecificInfo extends BaseDescriptor : bit(8)
tag=DecSpecificInfoTag
{
   // empty. To be filled by classes extending this class.
}
```

7.2.6.7.2 Semantics

The decoder specific information constitutes an opaque container with information for a specific media decoder. The existence and semantics of decoder specific information depends on the values of `DecoderConfigDescriptor.streamType` and `DecoderConfigDescriptor.objectTypeIndication`.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 14496-2 the syntax and semantics of decoder specific information are defined in Annex K of that part.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 14496-3 the syntax and semantics of decoder specific information are defined in subpart 1, subclause 1.6 of that part.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to scene description streams the semantics of decoder specific information is defined in ISO/IEC 14496-11.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 13818-7 the decoder specific information consists of an „adif_header()" and an access unit is a „raw_data_block()" as defined in ISO/IEC 13818-7.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 11172-3 or ISO/IEC 13818-3 the decoder specific information is empty since all necessary data is contained in the bitstream frames itself. The access units in this case are the „frame()" bitstream element as is defined in ISO/IEC 11172-3.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 10918-1, the decoder specific information is:

```
class JPEG_DecoderConfig extends DecoderSpecificInfo : bit(8)
tag=DecSpecificInfoTag {
   int(16) headerLength;
   int(16) Xdensity;
   int(16) Ydensity;
   int(8) numComponents;
}
```

with

`headerLength` –indicates the number of bytes to skip from the beginning of the stream to find the first pixel of the image.

`Xdensity` and `Ydensity` – specify the pixel aspect ratio.

`numComponents` – indicates whether the image has Y component only or is Y, Cr, Cb. It shall be equal to 1 or 3.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to interaction streams, the decoder specific information is:

```
class UIConfig extends DecoderSpecificInfo : bit(8) tag=DecSpecificInfoTag {
      bit(8) deviceNamelength;
      bit(8) deviceName[deviceNamelength];
      bit(8) devSpecInfo[sizeOfInstance – deviceNamelength - 1];
}
```

with

`deviceNameLength` –indicates the number of bytes in the deviceName field

`deviceName` –indicates the name of the class of device, which allows the terminal to invoke the appropriate interaction decoder.

`devSpecInfo` –is a opaque container with information for a device specific handler.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refers to extended BIFS configuration (0x04), the decoder specific information is:

```
class BIFSConfigEx extends DecoderSpecificInfo : bit (8) tag = DecSpecificInfoTag
{
   ExtendedBIFSConfig extBIFSConfig;
}

abstract aligned(8) expandable (..) class ExtendedBIFSConfig : bit (8) tag =
0x01..0xFF {
  //empty. To be filled by classes extending this class.
}
```

The class `BIFSConfigEx` contains an `ExtendedBIFSConfig`. `ExtendedBIFSConfig` is the base class for new classes ment to hold decoder specific info. With this approach, new BIFS streams will have streamType 3 and objectTypeIndication 0x04, but will use decoder configuration depending on the tag of the `ExtendedBIFSConfig`.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refers to AFX streams (0x05), the decoder specific information is:

```
class AFXConfig extends DecoderSpecificInfo : bit(8) tag=DecSpecificInfoTag {
   AFXExtDescriptor afxext;
}
abstract class AFXExtDescriptor extends BaseDescriptor : bit(8) tag = 0..100
{
}
```

`AFXExtDescriptor` is an abstract class used as a placeholder for an optional DecoderSpecificInfo defined in table "DecoderSpecificInfo for AFX streams" in ISO/IEC 14496-16. The tag refers to a specific node compression scheme as defined in table "AFX object code" in ISO/IEC 14496-16.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 15444-1, the decoder specific information is:

```
class JPEG2000_DecoderConfig extends DecoderSpecificInfo : bit(8)
tag=DecSpecificInfoTag {
   int(32) height;
   int(32) width;
   int(16) nc;
   int(8) BPC;
   int(8) C;
   int(8) UnkC;
   int(8) IPR;
}
```

The definition of the fields is extracted from ISO/IEC 15444-1 and is formulated as follows:

`height`: Image area height. The value of this parameter indicates the height of the image area. This field is stored as a 4-byte big endian unsigned integer.

`width`: Image area width. The value of this parameter indicates the width of the image area. This field is stored as a 4-byte big endian unsigned integer.

`nc`: Number of components. This parameter specifies the number of components in the codestream and is stored as a 2-byte big endian unsigned integer. The value of this field shall be equal to the value of the Csiz field in the SIZ marker in the codestream.

`BPC`: Bits per component. This parameter specifies the bit depth of the components in the codestream, minus 1, and is stored as a 1-byte field.

C: Compression type. This parameter specifies the compression algorithm used to compress the image data. The value of this field shall be 7. It is encoded as a 1-byte unsigned integer. Other values are reserved for ISO use.

UnkC: Colourspace Unknown. This field specifies if the actual colourspace of the image data in the codestream is known. This field is encoded as a 1-byte unsigned integer. Legal values for this field are 0, if the colourspace of the image is known and correctly specified in the Colourspace Specification boxes within the file, or 1, if the colourspace of the image is not known. A value of 1 will be used in cases such as the transcoding of legacy images where the actual colourspace of the image data is not known. In those cases, while the colourspace interpretation methods specified in the file may not accurately reproduce the image with respect to some original, the image should be treated as if the methods do accurately reproduce the image. Values other than 0 and 1 are reserved for ISO use.

IPR: Intellectual Property. This parameter indicates whether this JP2 file contains intellectual property rights information. If the value of this field is 0, this file does not contain rights information, and thus the file does not contain an IPR box. If the value is 1, then the file does contain rights information and thus does contain an IPR box as defined in I.6. Other values are reserved for ISO use.

The set of parameters defined above may all be extracted from the JP2 header box and are informal for setting up the JPEG 2000 decoder. However, if any conflict occurs with parameters from the JPEG 2000 header box in the Access Unit, the later have precedence.

7.2.6.8 SLConfigDescriptor

This descriptor defines the configuration of the sync layer header for this elementary stream. The specification of this descriptor is provided together with the specification of the sync layer in 7.3.2.3.

7.2.6.9 IP_IdentificationDataSet

7.2.6.9.1 Syntax

```
abstract class IP_IdentificationDataSet extends BaseDescriptor
     : bit(8) tag=ContentIdentDescrTag..SupplContentIdentDescrTag
{
  // empty. To be filled by classes extending this class.
}
```

7.2.6.9.2 Semantics

This class is an abstract base class that is extended by the descriptor classes that implement IP identification. A descriptor that allows to aggregate classes of type IP_IdentificationDataSet may actually aggregate any of the classes that extend IP_IdentificationDataSet.

7.2.6.10 ContentIdentificationDescriptor

7.2.6.10.1 Syntax

```
class ContentIdentificationDescriptor extends IP_IdentificationDataSet
     : bit(8) tag=ContentIdentDescrTag
{
  const bit(2)  compatibility=0;
  bit(1)    contentTypeFlag;
  bit(1)    contentIdentifierFlag;
  bit(1)    protectedContent;
  bit(3)    reserved = 0b111;
  if (contentTypeFlag)
    bit(8) contentType;
  if (contentIdentifierFlag) {
    bit(8) contentIdentifierType;
    bit(8) contentIdentifier[sizeOfInstance-2-contentTypeFlag];
  }
}
```

7.2.6.10.2 Semantics

The content identification descriptor is used to identify content. All types of elementary streams carrying content can be identified using this mechanism. The content types include audio, visual and scene description data. Multiple content identification descriptors may be associated to one elementary stream. These descriptors shall never be detached from the ES_Descriptor.

`compatibility` – must be set to 0.

`contentTypeFlag` – flag to indicate if a definition of the type of content is available.

`contentIdentifierFlag` – flag to indicate presence of creation ID.

`protectedContent` - if set to one indicates that the elementary streams that refer to this IP_IdentificationDataSet are protected by a method outside the scope of ISO/IEC 14496. The behavior of the terminal compliant with the ISO/IEC 14496 specifications when processing such streams is undefined.

`contentType` – defines the type of content using one of the values specified in Table 7.

Table 7 — contentType Values

0	Audio-visual
1	Book
2	Serial
3	Text
4	Item or Contribution (e.g. article in book or serial)
5	Sheet music
6	Sound recording or music video
7	Still Picture
8	Musical Work
9-254	Reserved for ISO use
255	Others

`contentIdentifierType` – defines a type of content identifier using one of the values specified in Table 8.

Table 8 — contentIdentifierType Values

0	ISAN	International Standard Audio-Visual Number
1	ISBN	International Standard Book Number
2	ISSN	International Standard Serial Number
3	SICI	Serial Item and Contribution Identifier
4	BICI	Book Item and Component Identifier
5	ISMN	International Standard Music Number
6	ISRC	International Standard Recording Code
7	ISWC-T	International Standard Work Code (Tunes)
8	ISWC-L	International Standard Work Code (Literature)
9	SPIFF	Still Picture ID
10	DOI	Digital Object Identifier
11-255	Reserved for ISO use	

`contentIdentifier` – international code identifying the content according to the preceding `contentIdentifierType`.

7.2.6.11 SupplementaryContentIdentificationDescriptor

7.2.6.11.1 Syntax

```
class SupplementaryContentIdentificationDescriptor extends
   IP_IdentificationDataSet : bit(8) tag= SupplContentIdentDescrTag
{
   bit(24)languageCode;
   bit(8) supplContentIdentifierTitleLength;
   bit(8) supplContentIdentifierTitle[supplContentIdentifierTitleLength];
   bit(8) supplContentIdentifierValueLength;
   bit(8) supplContentIdentifierValue[supplContentIdentifierValueLength];
}
```

7.2.6.11.2 Semantics

The supplementary content identification descriptor is used to provide extensible identifiers for content that are qualified by a language code. Multiple supplementary content identification descriptors may be associated to one elementary stream. These descriptors shall never be detached from the ES_Descriptor.

`language code` – This 24 bits field contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields.

`supplementaryContentIdentifierTitleLength` – indicates the length of the subsequent `supplementaryContentIdentifierTitle` in bytes.

`supplementaryContentIdentifierTitle` – identifies the title of a supplementary content identifier that may be used when a numeric content identifier (see 7.2.6.11) is not available.

`supplementaryContentIdentifierValueLength` – indicates the length of the subsequent `supplementaryContentIdentifierValue` in bytes.

`supplementaryContentIdentifierValue` – identifies the value of a supplementary content identifer associated to the preceding `supplementaryContentIdentifierTitle`.

7.2.6.12 IPI_DescrPointer

7.2.6.12.1 Syntax

```
class IPI_DescrPointer extends BaseDescriptor : bit(8) tag=IPI_DescrPointerTag {
   bit(16) IPI_ES_Id;
}
```

7.2.6.12.2 Semantics

The `IPI_DescrPointer` class contains a reference to the elementary stream that includes the `IP_IdentificationDataSets` that are valid for this stream. This indirect reference mechanism allows to convey such descriptors only in one elementary stream while making references to it from any `ES_Descriptor` that shares the same information.

ES_Descriptors for elementary streams that are intended to be accessible regardless of the availability of a referred stream shall explicitly include their IP_IdentificationDataSets instead of using an IPI_DescrPointer.

IPI_ES_Id – the ES_ID of the elementary stream whose ES_Descriptor contains the IP Information valid for this elementary stream. If the ES_Descriptor for IPI_ES_Id is not available, the IPI status of this elementary stream is undefined.

7.2.6.13 IPMP_DescriptorPointer

7.2.6.13.1 Syntax

```
class IPMP_DescriptorPointer extends BaseDescriptor :
bit(8) tag = IPMP_DescrPtrTag
{
     bit(8) IPMP_DescriptorID;
     if (IPMP_DescriptorID == 0xff){
     bit(16) IPMP_DescriptorIDEx;
     bit(16) IPMP_ES_ID;
  }
}
```

7.2.6.13.2 Semantics

The IPMP_DescriptorPointer appears in the ipmpDescPtr section of an OD or ESD structures.

The presence of this descriptor in an object descriptor indicates that all streams referred to by embedded ES_Descriptors are subject to protection and management by the IPMP System or IPMP Tool specified in the referenced IPMP_Descriptor.

The presence of this descriptor in an ES_Descriptor indicates that the stream associated with this descriptor is subject to protection and management by the IPMP System or IPMP Tool specified in the referenced IPMP_Descriptor.

The IPMP_DescriptorPointer supports the ability to identify which specific IPMP stream or streams the IPMP tools declared in the corresponding IPMP_Descriptor, identified by the IPMP_DescriptorIDEx, wish to receive. Multiple IPMP tools may receive updates from the same stream.

IPMP_DescriptorID is the ID of the IPMP_Descriptor being referred to. The bit(8) field is to support backward compatibility, for which support for extended IPMP stream association is not provided for.

IPMP_DescriptorIDEx is the ID of the IPMP_Descriptor being referred to. The bit(16) field refers to extension defined IPMP_Descriptors and also supporting the extended IPMP stream association.

IPMP_ES_ID is the id of an IPMP stream that may carry messages intended to the tool pointed to by IPMP_DescriptorIDEx. In case more than one IPMP stream is needed to feed the IPMP tool, several IPMP_DescriptorPointer can be given with the same IPMP_DescriptorIDEx and different IPMP_ES_ID. If the IPMP_ES_ID is null, it means the IPMP tool does not require an IPMP stream. A value of 2^16-1 for IPMP_ES_ID indicates that this field should be ignored, meaning that the tool pointed to by IPMP_DescriptorIDEx may receive messages from any IPMP stream within the presentation.

The list of IPMP streams identified by occurrences of the IPMP_ES_ID field (with a value different than 2^16-1) for a single IPMP_DescriptorIDEx is exhaustive: the IPMP tool identified by the IPMP_DescriptorIDEx may not receive messages from any other IPMP streams than the ones identified in this list. In order to facilitate editing, the IPMP_DescriptorPointer must be modified when stored in a file:

the IPMP_ES_ID field must be replaced with the corresponding index in the OD track's 'mpod' table as defined in ISO/IEC 14496-14.

7.2.6.14 IPMP Descriptor

7.2.6.14.1 Syntax

```
class IPMP_Descriptor() extends BaseDescriptor : bit(8) tag = IPMP_DescrTag
{
    bit(8) IPMP_DescriptorID;
    unsigned int(16) IPMPS_Type;
    if (IPMP_DescriptorID == 0xFF && IPMPS_Type == 0xFFFF){
        bit(16) IPMP_DescriptorIDEx;
        bit(128) IPMP_ToolID;
        bit(8) controlPointCode;
        if (controlPointCode > 0x00)
            bit(8) sequenceCode;
        IPMP_Data_BaseClass IPMPX_data[];
    }
    else if (IPMPS_Type == 0)
        bit(8) URLString[sizeOfInstance-3];
    else
        bit(8) IPMP_data[sizeOfInstance-3];
}
```

7.2.6.14.2 Semantics

The IPMP_Descriptor carries IPMP information for one or more IPMP System or IPMP Tool instances. It shall also contain optional instantiation information for one or more IPMP Tool instances.

IPMP_Descriptors are conveyed in either initial object descriptors, object descriptors or object descriptor streams via IPMP_DescriptorUpdate commands. Multiple definitions of the same IPMP_Descriptor within a single IPMP_DescriptorUpdate command or a single decoder specific information structure for an IPMP stream are not allowed. The behavior in such a situation is undefined. Note that, however, an IPMP_Descriptor may be modified/updated through subsequent IPMP_DescriptorUpdate commands received in the OD stream. IPMP_Descriptors shall be referenced by object descriptors or ES_Descriptors, using IPMP_DescriptorPointer.

IPMP_DescriptorID - a unique ID for this IPMP_Descriptor within its name scope. Values of "0x00" and "0xFF" are forbidden in the case of signaling an extension descriptor. The scope of the IPMP_DescriptorID is suggested to be the same as the OD, or IOD in which is it contained. IPMP_DescriptorID is for use in systems conforming to the previous definition as well as a signal indicating the use of IPMP_DescriptorIDEx for IPMP extensions.

Note 1: Although it is possible to implement an application supporting both the use of IPMP protection schemes defined through the use of IPMP_Descriptors some of which contain IPMP_DescriptorID and some of which contain IPMP_DescriptorIDEx to protect separate streams, the behavior of the association of a single stream to both types of IPMP_Descriptors is undefined.

IPMP_DescriptorIDEx - a unique ID for this IPMP_Descriptor within its name scope. Values of "0x0000" and "0xFFFF" are forbidden. The scope of the IPMP_DescriptorIDEx is suggested to be the same as the OD, or IOD in which is it contained.

IPMP_ToolID – the IPMP_ToolID of the IPMP Tool described by this IPMP_Descriptor. A zero, "0" value does not correspond to an IPMP Tool but is used to indicate the presence of a URL.

URLString[] - contains a UTF-8 encoded URL that shall point to the location of a remote IPMP_Descriptor. If the IPMPS_Type of this IPMP_Descriptor is 0, another URL is referenced. This process continues until an IPMP_Descriptor with a non-zero IPMPS_Type is accessed.

controlPointCode – specifies the IPMP control point at which the IPMP Tool resides, and is one of the following values:

controlPointCode	Description
0x00	No control point.
0x01	Control Point between the decode buffer and the decoder. This is between the decode buffer and class loader for MPEG-J streams.
0x02	Control Point between the decoder and the composition buffer.
0x03	Control Point between the composition buffer and the compositor.
0x04	BIFS Tree
0x05-0xDF	ISO Reserved
0xE0-0xFE	User defined
0xFF	Forbidden

Note 2: The only difference between receiving composition units before the CB and after the CB in the MPEG-4 Systems decoder model is the order in which the units are received when the associated DTS is different from the CTS; in this case the decoding order is different from the composition order. For example, suppose that a watermark payload is embedded in more than a single video frame; if the watermark payload was embedded in decoding order, it has to be extracted before the CB; instead, if it was embedded in composition order, it has to be extracted after the CB.

Note 3: For streams of type "0x01", ObjectDescriptor and of type "0x02", ClockReferenceStream, only a controlPointCode of "0x00", "0x01" or the range "0xE0-0xFE" are meaningful.

sequenceCode - The higher the sequence code, the higher the sequencing priority of the IPMP Tool instance at the given control point. Thus the tool with the highest sequenceCode for a given control point on a given stream shall process data first for that control point for that stream. Two tools shall not have the same sequence number at the same control point for the same stream.

IPMPX_data – The IPMP data that is extended from IPMP_Data_BaseClass, for the given IPMP tool.

IPMP_data – Data of unspecified format.

7.2.6.14.3 IPMP Tool List Specification

For each tool, this includes

1. IPMP Tool Identifier
2. Optional Parametric Description of the Tool.
3. Alternative Tools to the given Tool, any one of which replace the others without loss of functionality.

The Tool List shall be in the IOD, in an IPMP_ToolListDescriptor. Binary IPMP Tools are carried in separate elementary streams associated with the IOD. The specification of the syntax for the Tool List is as follows.

The IPMP_ToolListDescriptor conveys the list of IPMP tools required to access the content associated with the InitialObjectDescriptor in which it is described, and may include a list of alternate IPMP tools or parametric descriptions of tools required to access the content.

7.2.6.14.3.1 IPMP_ToolListDescriptor

This Subclause defines syntax and semantics for the carriage of a list of IPMP Tools required for the processing of the presentation.

7.2.6.14.3.1.1 Syntax

```
class IPMP_ToolListDescriptor extends BaseDescriptor :
   bit(8) tag= IPMP_ToolsListDescrTag
{
   IPMP_Tool ipmpTool[0 .. 255];
}
```

7.2.6.14.3.1.2 Semantics

`IPMP_Tool` – a class describing a logical IPMP Tool required to access the content.

7.2.6.14.3.2 IPMP_Tool

The IPMP Tool Identifier (or `IPMP_ToolID`) is 128-bits long, and shall contain a unique identification number for the IPMP Tool. A registration authority for IPMP Tools that use a unique ID is required. The registration authority shall maintain an optional association of the download URLs for various implementations of the given tool for various platforms. These platforms will be described to adequate detail using a structured representation. The `IPMP_ToolID` identifies a specific IPMP Tool (not a specific implementation of such a tool), unless in the reserved range for parametrically defined tools. Currently assigned 16-bit `IPMPS_Types` shall be directly mapped to a 128-bit ID by prepending with 112 zero bits; the RA will be initialized with such values. Specific values within this 128-bit space are reserved for indicating parametric tools, the bitstream, the terminal, and other special addresses. These values shall not be assigned to registered Tools.

Table 9 — Values of IPMP_ToolID

IPMP_ToolID	Semantics
0x0000	Forbidden
0x0001	Content
0x0002	Terminal
0x0003 - 0x2000	Reserved for ISO use
0x2001 - 0xFFFF	Carry over from 14496-1 RA
0x10000 - 0x100FF	Parametric Tools or Alternative Tools
0x100FF – 2^128-2	Open for registration
2^128-1	Forbidden

7.2.6.14.3.2.1 Syntax

```
class IPMP_Tool extends BaseDescriptor :
   bit(8) tag= IPMP_ToolTag
{
   bit(128)  IPMP_ToolID;
   bit(1)    isAltGroup;
   bit(1)    isParametric;
   const bit(6)     reserved=0b0000.00;

   if(isAltGroup){
     bit(8)    numAlternates;
```

```
      bit(128) specificToolID[numAlternates];
   }
   if(isParametric)
      IPMP_ParamtericDescription  toolParamDesc;
   ByteArray ToolURL[];
}
```

7.2.6.14.3.2.2 Semantics

Each instance of Class `IPMP_Tool` identifies one IPMP Tool that is required by the Terminal to Consume the Content. This Tool shall be specified either as a unique implementation, as one of a list of alternatives, or through a parametric description.

A unique implementation is indicated by the `isAltGroup` and `isParametric` fields both set to zero. In this case, the `IPMP_ToolID` shall be from the range reserved for specific implementations of an IPMP Tool and shall directly indicate the required Tool.

In all other cases, the `IPMP_ToolID` serves as a Content-specific abstraction for an IPMP Tool ID since the actual IPMP Tool ID of the Tool is not known at the time of authoring the Content, and will depend on the Terminal implementation at a given time for a given piece of Content.

A parametric description is indicated by setting the `isParametric` field to one. In this case, the Terminal shall select an IPMP Tool that meets the criteria specified in the following parametric description. In this case, the `IPMP_ToolID` shall be from the range reserved for Parametric Tools or Alternative Tools. The actual IPMP Tool ID of the Tool that the terminal implementation selects to fulfill this parametric description is known only to the Terminal. All the Content, and other tools, will refer to this Tool, for this Content, via the `IPMP_ToolID` specified. Note, this is not for message addressing.

A list of alternative Tools is indicated by setting the `isAltGroup` flag to ”1”. The subsequent specific Tool IDs indicate the Tools that are equivalent alternatives to each other. If the `isParametric` field is also set to one, any Tool that is selected under the conditions for parametric tools (as discussed in the paragraph above) shall be considered by the Terminal to be another equivalent alternative to those specified via specific Tool IDs. The Terminal shall choose one from these equivalent alternatives at its discretion. The actual IPMP Tool ID of this Tool is known only to the Terminal.

`IPMP_ToolID` – the identifier of the IPMP Tool, as discussed above.

`isAltGroup` – if set to one, this `IPMP_Tool` contains a list of alternate IPMP Tools.

`numAlternates` – the number of alternative IPMP Tools specified in `IPMP_Tool`.

`specificToolID` – an array of the IDs of specific alternative IPMP Tools that can allow consumption of the content.

`isParametric` – `IPMP_Tool` contains a parametric description of an IPMP Tool. In this case, `IPMP_ToolID` is an identifier for the parametrically described IPMP Tool, and the Terminal shall route information specified in the bitstream for `IPMP_ToolID` to the specific IPMP Tool instantiated by the terminal.

`ToolURL` – An array of informative URLs from which one or more tools specified in `IPMP_Tool` may be obtained in a manner defined outside the scope of these specifications.

7.2.6.14.3.3 IPMP_ParametricDescription

Using a parametric description, the content provider can now describe what type of IPMP tool is required to playback the content, instead of using fixed tool IDs. For example, the content provider can specify that an AES tool, with block size of 128 bits is required to decrypt video stream. The IPMP terminal, upon receiving such description specifying this tool, can then choose an optimised AES tool from the embedded tools.

This Subclause contains an illustration of the hierarchy that a parametric description would follow. It does not attempt to define any specific scheme for any specific Tool type. We anticipate that only a basic framework will appear in the current version of the specification, and enhancements to the same will be left for future addendums and/or versions.

1. Optional comment
2. Version of parametric description syntax
3. Class of Tool

   e.g. Decryption, Rights Language Parser
4. Sub-class of Tool
   a. e.g. for Decryption: AES, DES, NESSIE etc
   b. e.g. for Watermarking: “Panos’s watermarking tool” etc
   c. e.g. for Rights Language Parser: “Fred’s Rights Parser”
   d. e.g. for Protocol Parser: “Mary’s Protocol Parser”
5. Sub-class-specific information
   a. e.g. for DES: number of bits, stream and/or block decipher capability
   b. e.g. for Rights Language Parser : version

The parametric description is defined to allow a generic description of any type of IPMP tool, no matter the type of tool.

7.2.6.14.3.3.1 Syntax

```
class IPMP_ParamtericDescription extends IPMP_Data_BaseClass:
bit(8) tag = IPMP_ParamtericDescription_tag = 0x10
{
   ByteArray      descriptionComment;
   bit(8)      majorVersion;
   bit(8)      minorVersion;
   bit(32)     numOfDescriptions;
   For (int i = 0; i<numOfDescriptions; i++){
      ByteArray class;
      ByteArray subClass;
      ByteArray typeData;
      ByteArray type;
      ByteArray addedData;
   }
}
```

7.2.6.14.3.3.2 Semantics

`class` - class of the parametrically described tool, for example, decryption.

`subClass` - sub-class of the parametrically described tool, for example, AES under decryption class.

`typeData` - specific type data to describe a particular type of tool, for example, Block_length, to further specify a AES decryption tool.

`type` - value of the type data above, for example, 128 for the Block_length.

`addedData` - Any additional data which may help to further describe the parametrically defined tool.

7.2.6.14.3.4 ByteArray

This Subclause defines syntax and semantics to carry a generic string or array of bytes which is used extensively throughout the IPMP specifications.

7.2.6.14.3.4.1 Syntax

```
expandable class ByteArray
{
  bit(8) data[sizeOfInstance()];
}
```

7.2.6.14.3.4.2 Semantics

`data` - the string or array of bytes carried.

7.2.6.14.4 Implementation of a Registration Authority (RA)

CISAC will serve as the JTC 1 Registration Authority for the IPMPS_Type as defined in this Subclause. The Registration Authority shall execute its duties in compliance with Annex E of the JTC 1 Directives. The registered IPMPS_Type is hereafter referred to as the Registered Identifier (RID).

The Registration Management Group (RMG) will review appeals filed by organizations whose request for an RID to be used in conjunction with ISO/IEC 14496 has been denied by the Registration Authority.

Annex B provides information on the procedure for registering a unique IPMPS_Type value.

7.2.6.15 QoS_Descriptor

7.2.6.15.1 Syntax

```
class QoS_Descriptor extends BaseDescriptor : bit(8) tag=QoS_DescrTag {
  bit(8) predefined;
  if (predefined==0) {
    QoS_Qualifier qualifiers[];
  }
}
```

7.2.6.15.2 Semantics

The QoS_descriptor conveys the requirements that the ES has on the transport channel and a description of the traffic that this ES will generate. A set of predefined values is to be determined; customized values can be used by setting the `predefined` field to 0.

`predefined` – a value different from zero indicates a predefined QoS profile according to Table 10.

Table 10 — Predefined QoS Profiles

`predefined` value	**description**
0x00	Custom
0x01 - 0xff	Reserved

`qualifier` – an array of one or more `QoS_Qualifiers`.

7.2.6.15.3 QoS_Qualifier

7.2.6.15.3.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class QoS_Qualifier : bit(8) tag=0x01..0xff
{
   // empty. To be filled by classes extending this class.
}

class QoS_Qualifier_MAX_DELAY extends QoS_Qualifier : bit(8) tag=0x01 {
   unsigned int(32) MAX_DELAY;
}

class QoS_Qualifier_PREF_MAX_DELAY extends QoS_Qualifier : bit(8) tag=0x02 {
   unsigned int(32) PREF_MAX_DELAY;
}

class QoS_Qualifier_LOSS_PROB extends QoS_Qualifier : bit(8) tag=0x03 {
   double(32) LOSS_PROB;
}

class QoS_Qualifier_MAX_GAP_LOSS extends QoS_Qualifier : bit(8) tag=0x04 {
   unsigned int(32) MAX_GAP_LOSS;
}

class QoS_Qualifier_MAX_AU_SIZE extends QoS_Qualifier : bit(8) tag=0x41 {
   unsigned int(32) MAX_AU_SIZE;
}

class QoS_Qualifier_AVG_AU_SIZE extends QoS_Qualifier : bit(8) tag=0x42 {
   unsigned int(32) AVG_AU_SIZE;
}

class QoS_Qualifier_MAX_AU_RATE extends QoS_Qualifier : bit(8) tag=0x43 {
   unsigned int(32) MAX_AU_RATE;
}

class QoS_Qualifier_REBUFFERING_RATIO extends QoS_Qualifier : bit(8) tag=0x44 {
   bit(8) REBUFFERING_RATIO;
}
```

7.2.6.15.3.2 Semantics

QoS qualifiers are defined as derived classes from the abstract `QoS_Qualifier` class. They are identified by means of their class tag. Unused tag values up to and including 0x7F are reserved for ISO use. Tag values from 0x80 up to and including 0xFE are user private. Tag values 0x00 and 0xFF are forbidden.

`MAX_DELAY` – Maximum end to end delay for the stream in microseconds.

`PREF_MAX_DELAY` – Preferred end to end delay for the stream in microseconds.

`LOSS_PROB` – Allowable loss probability of any single AU as a fractional value between 0.0 and 1.0.

`MAX_GAP_LOSS` – Maximum allowable number of consecutively lost AUs.

`MAX_AU_SIZE` – Maximum size of an AU in bytes.

`AVG_AU_SIZE` – Average size of an AU in bytes.

`MAX_AU_RATE` – Maximum arrival rate of AUs in AUs/second.

`REBUFFERING_RATIO` – Ratio of the decoding buffer that should be filled in case of prebuffering or rebuffering. The ratio is expressed in percentage, with an integer value between 0 and 100. Values outside that range are reserved.

7.2.6.15.3.2.1 Rebuffering

In certain scenarios the System Decoder Model cannot be strictly observed. This is the case of e.g. file retrieval scenarios in which the data is pulled from a remote server over a network with unpredictable performances. In such a case prebuffering and/or rebuffering may be required in order to allow for a better quality in the user experience. Note that scenarios involving real time streaming servers do not fall in this category, since a streaming server presumably delivers content according to the appropriate timeline.

An elementary stream is prebuffered when the decoder waits until the decodingBuffer has been filled up to a certain threshold before starting fetching data from it.

An elementary stream is rebuffered when a decoder stops fetching data from the decodingBuffer and before resuming fetching data waits until that buffer has been filled again up to a certain threshold.

In order to inform a receiver whether a certain elementary stream requires prebuffering and/or rebuffering the `QoS_Qualifier_REBUFFERING_RATIO` qualifier can be included in the Elementary Stream Descriptor (see 7.2.6.15.3.1). By default, in the absence of such qualifier, an elementary stream does not require pre-buffering or rebuffering.

7.2.6.16 ExtensionDescriptor

7.2.6.16.1 Syntax

```
abstract class ExtensionDescriptor extends BaseDescriptor
: bit(8) tag = ExtensionProfileLevelDescrTag, ExtDescrTagStartRange ..
ExtDescrTagEndRange {
   // empty. To be filled by classes extending this class.
}
```

7.2.6.16.2 Semantics

This class is an abstract base class that may be extended for defining additional descriptors in future. The available range of class tag values allow ISO defined extensions as well as private extensions. A descriptor that allows to aggregate ExtensionDescriptor classes may actually aggregate any of the classes that extend ExtensionDescriptor. Extension descriptors may be ignored by a terminal that conforms to ISO/IEC 14496-1.

7.2.6.17 RegistrationDescriptor

The registration descriptor provides a method to uniquely and unambiguously identify formats of private data streams.

7.2.6.17.1 Syntax

```
class RegistrationDescriptor extends BaseDescriptor  : bit(8)
tag=RegistrationDescrTag {
   bit(32) formatIdentifier;
   bit(8) additionalIdentificationInfo[sizeOfInstance-4];
}
```

7.2.6.17.2 Semantics

formatIdentifier – is a value obtained from a Registration Authority as designated by ISO.

additionalIdentificationInfo – The meaning of additionalIdentificationInfo, if any, is defined by the assignee of that formatIdentifier, and once defined, shall not change.

The registration descriptor is provided in order to enable users of ISO/IEC 14496-1 to unambiguously carry elementary streams with data whose format is not recognized by ISO/IEC 14496-1. This provision will permit ISO/IEC 14496-1 to carry all types of data streams while providing for a method of unambiguous identification of the characteristics of the underlying private data streams.

In the following Subclause and Annex B, the benefits and responsibilities of all parties to the registration of private data format are outlined.

7.2.6.17.2.1 Implementation of a Registration Authority (RA)

ISO/IEC JTC 1/SC 29 shall issue a call for nominations from Member Bodies of ISO or National Committees of IEC in order to identify suitable organizations that will serve as the Registration Authority for the formatIdentifier as defined in this Subclause. The selected organization shall serve as the Registration Authority. The so-named Registration Authority shall execute its duties in compliance with Annex E of the JTC 1 Directives. The registered private data formatIdentifier is hereafter referred to as the Registered Identifier (RID).

Upon selection of the Registration Authority, JTC 1 shall require the creation of a Registration Management Group (RMG) which will review appeals filed by organizations whose request for an RID to be used in conjunction with ISO/IEC 14496-1 has been denied by the Registration Authority.

Annex B provides information on the procedure for registering a unique format identifier.

7.2.6.18 Object Content Information Descriptors

7.2.6.18.1 Overview

This Subclause defines the descriptors that constitute the object content information. These descriptors may either be included in an OCI_Event in an OCI stream or be part of an object descriptor or ES_Descriptor as defined in 7.2.6.

7.2.6.18.2 OCI_Descriptor Class

7.2.6.18.2.1 Syntax

```
abstract class OCI_Descriptor extends BaseDescriptor
   : bit(8) tag= OCIDescrTagStartRange .. OCIDescrTagEndRange
{
   // empty. To be filled by classes extending this class.
}
```

7.2.6.18.2.2 Semantics

This class is an abstract base class that is extended by the classes specified in the subsequent Clauses. A descriptor or an OCI_Event that allows to aggregate classes of type OCI_Descriptor may actually aggregate any of the classes that extend OCI_Descriptor.

7.2.6.18.3 Content classification descriptor

7.2.6.18.3.1 Syntax

```
class ContentClassificationDescriptor extends OCI_Descriptor
            : bit(8) tag= ContentClassificationDescrTag {
  bit(32) classificationEntity;
  bit(16) classificationTable;
  bit(8) contentClassificationData[sizeOfInstance-6];
}
```

7.2.6.18.3.2 Semantics

The content classification descriptor provides one or more classifications of the event information. The `classificationEntity` field indicates the organization that classifies the content. The possible values have to be registered with a registration authority to be identified.

`classificationEntity` – indicates the content classification entity. The values of this field are to be defined by a registration authority to be identified.

`classificationTable` – indicates which classification table is being used for the corresponding classification. The classification is defined by the corresponding classification entity. 0x00 is a reserved value.

`contentClassificationData[]` – this array contains a classification data set using a non-default classification table.

7.2.6.18.4 Key Word Descriptor

7.2.6.18.4.1 Syntax

```
class KeyWordDescriptor extends OCI_Descriptor : bit(8) tag=KeyWordDescrTag {
  int i;
  bit(24) languageCode;
  bit(1) isUTF8_string;
  aligned(8) unsigned int(8) keyWordCount;
  for (i=0; i<keyWordCount; i++) {
    unsigned int(8) keyWordLength[[i]];
    if (isUTF8_string) then {
      bit(8) keyWord[[i]][keyWordLength[i]];
    } else {
      bit(16) keyWord[[i]][keyWordLength[i]];
    }
  }
}
```

7.2.6.18.4.2 Semantics

The key word descriptor allows the OCI creator/provider to indicate a set of key words that characterize the content. The choice of the key words is completely free but each time the key word descriptor appears, all the key words given are for the language indicated in `languageCode`. This means that, for a certain event, the key word descriptor must appear as many times as the number of languages for which key words are to be provided.

`languageCode` – contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`keyWordCount` – indicates the number of key words to be provided.

`keyWordLength` – specifies the length in characters of each key word.

`keyWord[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the key word.

7.2.6.18.5 Rating Descriptor

7.2.6.18.5.1 Syntax

```
class RatingDescriptor extends OCI_Descriptor : bit(8) tag=RatingDescrTag {
   bit(32) ratingEntity;
   bit(16) ratingCriteria;
   bit(8)  ratingInfo[sizeOfInstance-6];
}
```

7.2.6.18.5.2 Semantics

This descriptor gives one or more ratings, originating from corresponding rating entities, valid for a specified country. The `ratingEntity` field indicates the organization which is rating the content. The possible values have to be registered with a registration authority to be identified. This registration authority shall make the semantics of the rating descriptor publicly available.

`ratingEntity` – indicates the rating entity. The values of this field are to be defined by a registration authority to be identified.

`ratingCriteria` – indicates which rating criteria are being used for the corresponding rating entity. The value 0x00 is reserved.

`ratingInfo[]` – this array contains the rating information.

7.2.6.18.6 Language Descriptor

7.2.6.18.6.1 Syntax

```
class LanguageDescriptor extends OCI_Descriptor : bit(8) tag=LanguageDescrTag {
   bit(24) languageCode;
}
```

7.2.6.18.6.2 Semantics

This descriptor identifies the language of the corresponding audio/speech or text object that is being described.

`languageCode` – contains the ISO 639-2:1998 bibliographic three character language code of the corresponding audio/speech or text object that is being described.

7.2.6.18.7 Short Textual Descriptor

7.2.6.18.7.1 Syntax

```
class ShortTextualDescriptor extends OCI_Descriptor : bit(8)
tag=ShortTextualDescrTag {
  bit(24) languageCode;
  bit(1) isUTF8_string;
  aligned(8) unsigned int(8) nameLength;
  if (isUTF8_string) then {
    bit(8) eventName[nameLength];
    unsigned int(8) textLength;
    bit(8) eventText[textLength];
  } else {
    bit(16) eventName[nameLength];
    unsigned int(8) textLength;
    bit(16) eventText[textLength];
  }
}
```

7.2.6.18.7.2 Semantics

The short textual descriptor provides the name of the event and a short description of the event in text form.

`languageCode` – contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`nameLength` – specifies the length in characters of the event name.

`eventName[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the event name.

`textLength` – specifies the length in characters of the following text describing the event.

`eventText[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the text description for the event.

7.2.6.18.8 Expanded Textual Descriptor

7.2.6.18.8.1 Syntax

```
class ExpandedTextualDescriptor extends OCI_Descriptor : bit(8)
tag=ExpandedTextualDescrTag {
  int i;
  bit(24) languageCode;
  bit(1) isUTF8_string;
  aligned(8) unsigned int(8) itemCount;
  for (i=0; i<itemCount; i++){
    unsigned int(8) itemDescriptionLength[[i]];
    if (isUTF8_string) then {
      bit(8) itemDescription[[i]][itemDescriptionLength[i];
    } else {
      bit(16) itemDescription[[i]][itemDescriptionLength[i]];
    }
    unsigned int(8) itemLength[[i]];
    if (isUTF8_string) then {
         bit(8) itemText[[i]][itemLength[i]];
      } else {
         bit(16) itemText[[i]][itemLength[i]];
      }
   }
   unsigned int(8) textLength;
   int nonItemTextLength=0;
   while( textLength == 255 ) {
      nonItemTextLength += textLength;
      bit(8) textLength;
   }
   nonItemTextLength += textLength;
   if (isUTF8_string) then {
      bit(8) nonItemText[nonItemTextLength];
   } else {
      bit(16) nonItemText[nonItemTextLength];
   }
}
```

7.2.6.18.8.2 Semantics

The expanded textual descriptor provides a detailed description of an event, which may be used in addition to, or independently from, the short event descriptor. In addition to direct text, structured information in terms of pairs of description and text may be provided. An example application for this structure is to give a cast list, where for example the item description field might be "Producer" and the item field would give the name of the producer.

`languageCode` - contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`itemCount` – specifies the number of items to follow (itemised text).

`itemDescriptionLength` – specifies the length in characters of the item description.

`itemDescription[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the item description.

`itemLength` – specifies the length in characters of the item text.

`itemText[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the item text.

`textLength` – specifies the length in characters of the non itemised expanded text. The value 255 is used as an escape code, and it is followed by another `textLength` field that contains the length in bytes above 255. For lengths greater than 511 a third field is used, and so on.

`nonItemText[]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the non itemised expanded text.

7.2.6.18.9 Content Creator Name Descriptor

7.2.6.18.9.1 Syntax

```
class ContentCreatorNameDescriptor extends OCI_Descriptor
            : bit(8) tag= ContentCreatorNameDescrTag {
  int i;
  unsigned int(8) contentCreatorCount;
  for (i=0; i<contentCreatorCount; i++){
    bit(24) languageCode[[i]];
    bit(1) isUTF8_string[[i]];
    aligned(8) unsigned int(8) contentCreatorLength[[i]];
    if (isUTF8_string[[i]]) then {
      bit(8) contentCreatorName[[i]][contentCreatorLength[i]];
    } else {
      bit(16) contentCreatorName[[i]][contentCreatorLength[i]];
    }
  }
}
```

7.2.6.18.9.2 Semantics

The content creator name descriptor indicates the name(s) of the content creator(s). Each content creator name may be in a different language.

`contentCreatorCount` – indicates the number of content creator names to be provided.

`languageCode` – contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode (O/IEC 10646-1).

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`contentCreatorLength[[i]]` – specifies the length in characters of each content creator name.

`contentCreatorName[[i]][]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the content creator name.

7.2.6.18.10 Content Creation Date Descriptor

7.2.6.18.10.1 Syntax

```
class ContentCreationDateDescriptor extends OCI_Descriptor
            : bit(8) tag= ContentCreationDateDescrTag {
  bit(40) contentCreationDate;
}
```

7.2.6.18.10.2 Semantics

This descriptor identifies the date of the content creation.

`contentCreationDate` – contains the content creation date of the data corresponding to the event in question, in Universal Time, Co-ordinated (UTC) and Modified Julian Date (MJD) (see Annex A). This field is coded as 16 bits giving the 16 least significant bits of MJD followed by 24 bits coded as 6 digits in 4-bit Binary Coded Decimal (BCD). If the content creation date is undefined all bits of the field are set to 1.

7.2.6.18.11 OCI Creator Name Descriptor

7.2.6.18.11.1 Syntax

```
class OCICreatorNameDescriptor extends OCI_Descriptor
            : bit(8) tag=OCICreatorNameDescrTag {
  int i;
  unsigned int(8) OCICreatorCount;
  for (i=0; i<OCICreatorCount; i++) {
    bit(24) languageCode[[i]];
    bit(1) isUTF8_string;
    aligned(8) unsigned int(8) OCICreatorLength[[i]];
    if (isUTF8_string) then {
      bit(8) OCICreatorName[[i]][OCICreatorLength];
     } else {
      bit(16) OCICreatorName[[i]][OCICreatorLength];
    }
  }
}
```

7.2.6.18.11.2 Semantics

The name of OCI creators descriptor indicates the name(s) of the OCI description creator(s). Each OCI creator name may be in a different language.

`OCICreatorCount` – indicates the number of OCI creators.

`languageCode[[i]]` – contains the ISO 639-2:1998 bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`OCICreatorLength[[i]]` – specifies the length in characters of each OCI creator name.

`OCICreatorName[[i]]` – a Unicode (ISO/IEC 10646-1) encoded string that specifies the OCI creator name.

7.2.6.18.12 OCI Creation Date Descriptor

7.2.6.18.12.1 Syntax

```
class OCICreationDateDescriptor extends OCI_Descriptor
            : bit(8) tag=OCICreationDateDescrTag {
  bit(40) OCICreationDate;
}
```

7.2.6.18.12.2 Semantics

This descriptor identifies the creation date of the OCI description.

`OCICreationDate` - This 40-bit field contains the OCI creation date for the OCI data corresponding to the event in question, in Co-ordinated Universal Time (UTC) and Modified Julian Date (MJD) (see Annex A). This field is coded as 16 bits giving the 16 least significant bits of MJD followed by 24 bits coded as 6 digits in 4-bit Binary Coded Decimal (BCD). If the OCI creation date is undefined all bits of the field are set to 1.

7.2.6.18.13 SMPTE Camera Position Descriptor

7.2.6.18.13.1 Syntax

```
class SmpteCameraPositionDescriptor extends OCI_Descriptor : bit (8)
tag=SmpteCameraPositionDescrTag {
  unsigned int (8) cameraID;
  unsigned int (8) parameterCount;
  for (i=0; i<parameterCount; i++) {
    bit (8) parameterID;
    bit (32) parameter;
  }
}
```

7.2.6.18.13.2 Semantics

The SMPTE metadata descriptor provides metadata defined by the Proposed SMPTE Standard 315M of “camera positioning information conveyed by ancillary data packets.” The SMPTE 315M defines IDs and data formats for the following parameters:

- camera relative position
- camera pan
- camera tilt
- camera roll
- origin of world coordinate longitude
- origin of world coordinate latitude
- origin of world coordinate altitude
- vertical angle of view
- focus distance
- lens opening (iris or F-value)
- time address information
- object relative position

`cameraID` - contains the b(0-7) of C-ID of the UDW in Figure 6.

`parameterCount` - specifies the number of parameters and is equal to (the Data Count Word (DC) – 18) / 5.

`parameterID` - contains the b(0-7) of i-th IDn of the UDW.

`parameter` - contains the i-th Parameter n of the UDW (b(0-7) of each word).

7.2.6.18.13.3 Packet structure defined by SMPTE 315M

Ancillary data packet and space format is defined by ANSI/SMPTE 291M. The SMPTE 315M is one of the registered formats for a specific application of user data space defined by the 291M. The structure of binary-type camera positioning data packets described in the SMPTE 315M is illustrated in Figure 6.

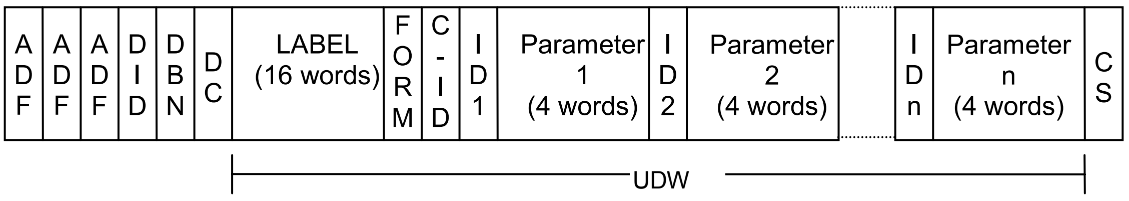


Figure 6 — Binary-type camera positioning data packets (SMPTE 315M)

Ancillary data is defined as 10-bit words. B(0-7), b8 and b9 represent actual data, even parity for b(0-7) and not b8 respectively except ADF.

ADF:	Ancillary Data Flag (000 h, 3ff h, 3ff h)
DID:	Data Identification Word (2f0 h)
DBN:	Data Block Number Word
DC:	Data Count Word
UDW:	User Data Words (up to 255 words)
LABEL:	SMPTE label for metadata of class “camera positioning information” (16 words)
FORM:	Data Type Identification Flag Word (1 word)
C-ID:	Camera Identification Word (1 word)
IDn:	Parameter Identification Word (1 word for each parameter)
Parameter n:	Parameter Data Words (4 words for each parameter)
CS:	Checksum Word

The 4 words LABEL(8-11) of LABEL(0-15) shall be set to ‘C’, ‘A’, ‘P’, ‘O’. The Data Type Identification Flag Word (FORM) indicates the data type of the camera identification word (C-ID), parameter identification word (IDn) and parameter data word (Parameter n) contained in the packet. In case of binary-type camera positioning data FORM(0-1) shall be set to 0 h.

7.2.6.18.14 Segment Descriptor

7.2.6.18.14.1 Syntax

```
class SegmentDescriptor extends OCI_Descriptor : bit(8) tag=SegmentDescriptorTag
{
   double  start;
   double  duration;
   bit(8)  segmentNameLength;
   bit(8)  segmentName [segmentNameLength];
};
```

7.2.6.18.14.2 Semantics

The segment descriptor defines labeled segments within a media stream with respect to the media time line. A segment for a given media stream is declared by conveying a segment descriptor with appropriate values as part of the object descriptor that declares that media stream. Conversely, when a segment descriptor exists in an object descriptor, it refers to all the media streams in that object descriptor. Segments can be referenced from the scene description through **url** fields of media nodes.

In order to use segment descriptors for the declaration of segments within a media stream, the notion of a media time line needs to be established. The media time line of a media stream may be defined through use of media time descriptor (see 7.2.6.18.15.1). In the absence of such explicit definitions, media time of the first composition unit of a media stream is assumed to be zero. In applications where random access into a media stream is supported, the media time line is undefined unless the media time descriptor mechanism is used.

`start` – specifies the media time (in seconds) of the start of the segment within the media stream.

`duration` – specifies the duration of the segment in seconds. A negative value denotes an infinite duration.

`SegmentNameLength` – the length of the `segmentName` field in characters.

`segmentName` – a Unicode [3] encoded string that labels the segment. The first character of the `segmentName` shall be an alphabetic character. The other characters may be alphanumeric, _, -, or a space character.

7.2.6.18.15 MediaTimeDescriptor

7.2.6.18.15.1 Syntax

```
class MediaTimeDescriptor extends OCI_Descriptor : bit(8) tag=MediaTimeDescrTag {
   double  mediaTimeStamp;
};
```

7.2.6.18.15.2 Semantics

The media time descriptor conveys a media time stamp. The descriptor establishes the mapping between the object time base and the media time line of a media stream. This descriptor shall only be conveyed within an OCI stream. The `startingTime`, `absoluteTimeFlag` and `duration` fields of the OCI event carrying this descriptor shall be set to 0. The association between the OCI stream and the corresponding media stream is defined by an object descriptor that aggregates ES descriptors for both of them (see 7.2).

`mediaTimeStamp` – a time stamp indicating the media time (MT, in seconds) of the associated media stream corresponding to the composition time (CT) of the access unit conveying the media time descriptor. Media time values $MT(AU_n)$ of other access units of the media stream can be calculated from the composition time $CT(AU_n)$ for that access unit as follows:

$$MT(AU_n) = CT(AU_n) - CT + MT$$

with MT and CT being the `mediaTimeStamp` and `compositionTimeStamp` (converted to seconds) values, respectively, for the access unit conveying the media time descriptor.

Note – When media time descriptor is used to associate a media time line with a media stream, the notion of “media time zero” does not necessarily correspond to the notion of “beginning of the stream”.

7.2.6.19 Extension Profile Level Descriptor

7.2.6.19.1 Syntax

```
class ExtensionProfileLevelDescriptor() extends ExtensionDescriptor :  bit(8)
ExtensionProfileLevelDescrTag {
      bit(8) profileLevelIndicationIndex;
      bit(8) ODProfileLevelIndication;
      bit(8) sceneProfileLevelIndication;
      bit(8) audioProfileLevelIndication;
      bit(8) visualProfileLevelIndication;
      bit(8) graphicsProfileLevelIndication;
```

```
    bit(8) MPEGJProfileLevelIndication;
    bit(8) TextProfileLevelIndication;
    bit(8) 3DCProfileLevelIndication;
}
```

7.2.6.19.2 Semantics

The `ExtensionProfileLevelDescriptor` conveys profile and level extension information. This descriptor is used to signal a profile and level indication set and its unique index and can be extended by ISO to signal any future set of profiles and levels.

`profileLevelIndicationIndex` – a unique identifier for the set of profile and level indications described in this descriptor within the name scope defined by the IOD.

`ODProfileLevelIndication` – an indication of the profile and level required to process object descriptor streams associated with the `InitialObjectDescriptor` containing this Extension Profile and Level descriptor.

`sceneProfileLevelIndication` – an indication of the profile and level required to process the scene graph nodes within scene description streams associated with the `InitialObjectDescriptor` containing this Extension Profile and Level descriptor.

`audioProfileLevelIndication` – an indication of the profile and level required to process audio streams associated with the `InitialObjectDescriptor` containing this Extension Profile and Level descriptor.

`visualProfileLevelIndication` – an indication of the profile and level required to process visual streams associated with the `InitialObjectDescriptor` containing this Extension Profile and Level descriptor.

`graphicsProfileLevelIndication` – an indication of the profile and level required to process graphics nodes within scene description streams associated with the `InitialObjectDescriptor` containing this Extension Profile and Level descriptor.

`MPEGJProfileLevelIndication` – an indication as defined in Table 11 of the MPEG-J profile and level required to process the content associated with the InitialObjectDescriptor containing this Extension Profile and Level descriptor.

Table 11 — MPEGJProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Personal profile	L1
0x02	Main profile	L1
0x03-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no MPEG-J profile specified	-
0xFF	no MPEG-J capability required	-
Note Usage of the value 0xFE may indicate that the content described by this InitialObjectDescriptor does not comply to any conformance point specified in ISO/IEC 14496-1		

`TextProfileLevelIndication` – an indication as defined in Table 12, of the Text Profile and Level specified in ISO/IEC 14496-18 and required to process the content associated with the InitialObjectDescriptor containing this Text Profile and Level descriptor.

Table 12 — TextProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Simple Text profile	L1
0x02	Simple Text profile	L2
0x03	Simple Text profile	L3
0x04	Advanced Simple Text profile	L1
0x05	Advanced Simple Text profile	L2
0x06	Advanced Simple Text profile	L3
0x07	Main Text profile	L1
0x08	Main Text profile	L2
0x09	Main Text profile	L3
0x0A-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no Text profile specified	-
0xFF	no text rendering capability required	-
Note: Usage of the value 0xFE may indicate that the content described by this descriptor does not comply to any conformance point specified in ISO/IEC 14496-18.		

`3DCProfileLevelIndication` – an indication of the 3D Compression Profile and Level specified in ISO/IEC 14496-16 and required to process the content associated with the InitialObjectDescriptor containing this 3D Compression Profile and Level descriptor

7.2.6.20 Profile Level Indication Index Descriptor

7.2.6.20.1 Syntax

```
class ProfileLevelIndicationIndexDescriptor () extends BaseDescriptor
: bit(8) ProfileLevelIndicationIndexDescrTag {
   bit(8) profileLevelIndicationIndex;
}
```

7.2.6.20.2 Semantics

`profileLevelIndicationIndex` – a unique identifier for the set of profile and level indications described in this descriptor within the name scope defined by the IOD.

7.2.7 Rules for Usage of the Object Description Framework

7.2.7.1 Aggregation of Elementary Stream Descriptors in a Single Object Descriptor

7.2.7.1.1 Overview

An object descriptor shall aggregate the descriptors for the set of elementary streams that is intended to be associated to a single node of the scene description and that usually relate to a single audio-visual object. The set of streams may convey a scaleable content representation as well as multiple alternative content representations, e.g., multiple qualities or different languages. Additional streams with IPMP and object content information may be attached.

These options are described by the ES_Descriptor syntax elements `streamDependenceFlag`, `dependsOn_ES_ID`, as well as `streamType`. The semantic rules for the aggregation of elementary stream descriptors within one object descriptor (OD) are specified in this Subclause.

7.2.7.1.2 Aggregation of Elementary Streams with the same streamType

An OD may aggregate multiple ES_Descriptors with the same `streamType` of either visualStream, audioStream or SceneDescriptionStream. However, descriptors for streams with two of these types shall not be mixed within one OD.

7.2.7.1.3 Aggregation of Elementary Streams with Different streamTypes

In the following cases ESs with different `streamType` may be aggregated:

- An OD may aggregate zero or one additional ES_Descriptor with `streamType` = ObjectContentInfoStream (see 7.2.4.2). This ObjectContentInfoStream shall be valid for the content conveyed through the other visual, audio or scene description streams whose descriptors are aggregated in this OD.
- An OD may aggregate zero or one additional ES_Descriptors with `streamType` = ClockReferenceStream (see 7.3.2.5). This ClockReferenceStream shall be valid for the ES within the name scope that refer to the ES_ID of this ClockReferenceStream in their SLConfigDescriptor.
- An OD may aggregate zero or more additional ES_Descriptors with `streamType` = IPMPStream (see 7.2.3.2). This IPMPStream shall be valid for the content conveyed through the other visual, audio or scene description streams whose descriptors are aggregated in this OD.

7.2.7.1.4 Aggregation of scene description streams and object descriptor streams

An object descriptor that aggregates one or more ES_Descriptors of `streamType` = SceneDescriptionStream may aggregate any number of additional ES_Descriptors with `streamType` = ObjectDescriptorStream. ES_Descriptors of `streamType` = ObjectDescriptorStream shall not be aggregated in object descriptors that do not contain ES_Descriptors of `streamType` = SceneDescriptionStream.

This means that scene description and object descriptor streams are always combined within one object descriptor. The dependencies between these streams are defined in 7.2.7.1.5.2.

7.2.7.1.5 Elementary Stream Dependencies

7.2.7.1.5.1 Independent elementary streams

ES_Descriptors within one OD with the same `streamType` of either audioStream, visualStream or SceneDescriptionStream that have `streamDependenceFlag=0` refer to independent elementary streams. Such independent elementary streams shall convey alternative representations of the same content. Only one of these representations shall be selected for use in the scene.

NOTE — Independent ESs should be ordered within an OD according to the content creator's preference. The ES that is first in the list of ES aggregated to one object descriptor should be preferable over an ES that follows later. In case of audio streams, however, the selection should for obvious reasons be done according to the prefered language of the receiving terminal.

7.2.7.1.5.2 Dependent elementary streams

ES_Descriptors within one OD with the same `streamType` of either audioStream, visualStream, SceneDescriptionStream or ObjectDescriptorStream that have `streamDependenceFlag=1` refer to dependent elementary streams. The ES_ID of the stream on which the dependent elementary stream depends is indicated by `dependsOn_ES_ID`. The ES_Descriptor with this ES_ID shall be aggregated to the same OD. One independent elementary stream per object descriptor and all its dependent elementary streams may be selected for concurrent use in the scene.

Stream dependencies are governed by the following rules:

- For dependent ES of `streamType` equal to either audioStream or visualStream the dependent ES shall have the same `streamType` as the ES on which it depends. This implies that the dependent stream contains enhancement information to the one it depends on. The precise semantic meaning of the dependencies is opaque at this layer.

- An ES with a `streamType` of SceneDescriptionStream shall only depend on an ES with `streamType` of SceneDescriptionStream or ObjectDescriptorStream.

— Dependency on an ObjectDescriptorStream implies that the ObjectDescriptorStream contains the object descriptors that are refered to by this SceneDescriptionStream.

— Dependency on a SceneDescriptionStream implies that the dependent stream contains enhancement information to the one it depends on. The dependent SceneDescriptionStream shall depend on the same ObjectDescriptorStream on which the other SceneDescriptionStream depends.

- An ES with a streamType of ObjectDescriptionStream shall only depend on an ES with streamType of SceneDescriptionStream or ObjectDescriptorStream.

— Dependency on a SceneDescriptorStream implies that there shall be one or more ESs with a streamType of SceneDescriptionStream depending on this ObjectDescriptorStream.

— Dependency on an ObjectDescriptionStream implies that the dependent stream contains additional object descriptors comprising the presentation described by SceneDescriptionStreams which are aggregated in the same object descriptor.

- An ES that flows upstream, as indicated by `DecoderConfigDescriptor.upStream` = 1 shall always depend upon another ES that has the `upStream` flag set to zero. This implies that this upstream is associated to the downstream it depends on. If the downstream is an ObjectDescriptorStream or SceneDescriptionStream, the upstream shall be associated to all downstreams specified in that ObjectDescriptorStream or SceneDescriptionStream.

- The availability of the dependent stream is undefined if an ES_Descriptor for the stream it depends upon is not available.

7.2.7.2 Linking Scene Description and Object Descriptors

7.2.7.2.1 Associating Object Descriptors to BIFS Nodes

Some BIFS nodes contain an **url** field. Such nodes are associated to their elementary stream resources (if any) via an object descriptor. The association is established by means of the `objectDescriptorID`, as specified in ISO/IEC 14496-11. The name scope for this ID is specified in 7.2.7.2.4.

Each BIFS node requires a specific streamType (audio, visual, inlined scene description, etc.) for its associated elementary streams. The associated object descriptor shall contain ES_Descriptors with this streamType. The behavior of the terminal is undefined if an object descriptor contains ES_Descriptors with stream types that are incompatible with the associated BIFS node.

Note that commands adding or removing object descriptors need not be co-incident in time with the addition or removal of BIFS nodes in the scene description that refer to such an object descriptor. However, the behavior of the terminal is undefined if a BIFS node in the scene description references an object descriptor that is no longer valid.

At times that the object descriptor is not available at the terminal, the terminal shall behave as if the the URL referencing the object descriptor was empty. In the case of visual streams for which the object descriptor has been deleted, the terminal shall render the last composition unit in the scene.

7.2.7.2.2 Multiple scene description and object description streams

An object descriptor that is associated to an **Inline** node of the scene description or that represents the primary access to content compliant with the ISO/IEC 14496 specifications (initial object descriptor) aggregates as a minimum, one scene description stream and the corresponding object descriptor stream (if additional elementary streams need to be referenced).

However, it is permissible to split both the scene description and the object descriptors in multiple streams. This allows a bandwidth-scaleable encoding of the scene description. Each stream shall contain a valid sequence of access units as defined in ISO/IEC 14496-11, and 7.2.5.2, respectively. All resulting scene description streams and object descriptor streams shall remain aggregated in a single object descriptor. The dependency mechanism shall be used to indicate how the streams depend on each other.

All streams shall continue to be processed by a single scene description and object descriptor decoding process, respectively. The time stamps of the access units in different streams shall be used to re-establish the original order of access units.

NOTE — This form of partitioning of the scene description and the object descriptor streams in multiple streams is not visible in the scene description itself.

7.2.7.2.3 Scene and Object Description in Case of Inline Nodes

The BIFS scene description allows to recursively partition a scene through the use of **Inline** nodes (see ISO/IEC 14496-11). Each **Inline** node is associated to an object descriptor that points to at least one additional scene description stream as well as another object descriptor stream (if additional elementary streams need to be referenced). An example for such a hierarchical scene description can be found in 7.2.7.3.8.2.

7.2.7.2.4 Name Scope of Identifiers

The scope of the `objectDescriptorID, ES_ID` and `IPMP_DescriptorID` identifiers that label the object descriptors, elementary stream descriptors and IPMP descriptors, respectively, is defined as follows. This definition is based on the restriction that associated scene description and object descriptor streams shall always be aggregated in a single object descriptor, as specified in 7.2.7.1.4. The following rule defines the name scope:

- Two scene related identifiers (`objectDescriptorID`, `nodeID , ROUTEID or protoID`) belong to the same name scope if and only if these identifiers occur in elementary streams with a `streamType` of either ObjectDescriptorStream or SceneDescriptionStream that are aggregated in a single initial object descriptor or a single object descriptor associated to an **Inline** node.

- Two stream related identifiers (ES_ID or IPMP_DescriptorID) belong to the same name scope if and only if these identifiers relate to streams that are attached to the same communication session that is established as described in 7.2.7.3.6.

NOTE 1 — Hence, the difference between the two methods specified in 7.2.7.2.2 and 7.2.7.2.3 above to partition a scene description in multiple streams is that the first method allows multiple scene description streams that refer to the same name scope while an **Inline** node opens a new name scope.

NOTE 2 — This implies that a URL in an object descriptor opens a new name scope since it points to an object descriptor that is not carried in the same ObjectDescriptorStream.

7.2.7.2.5 Reuse of identifiers

Within a single name scope an ES_ID identifier shall always refer to a single instance of an elementary stream.

Note: If two ES_Descriptors within two object descriptors reference a given ES_ID, this means that the second reference may not receive the stream content from the beginning if the first reference has already started the stream.

For reasons of error resilience, it is recommended not to reuse `objectDescriptorID` and `ES_ID` identifiers to identify more than one object or elementary stream, respectively, within one presentation. That means, if an object descriptor or elementary stream descriptor is removed by means of an OD command and later on reinstalled with another OD command, then it shall still point to the same content item as before.

7.2.7.3 ISO/IEC 14496 Content Access

7.2.7.3.1 Introduction

In order to access ISO/IEC 14496 compliant content it is a pre-condition that an initial object descriptor to such content is known through means outside the scope of ISO/IEC 14496. The subsequent content access procedure is specified conceptually, using a number of walk throughs. Its precise definition depends on the chosen delivery layer.

For applications that implement the DMIF Application Interface (DAI) specified in ISO/IEC 14496-6 which abstracts the delivery layer, a mapping of the conceptual content access procedure to calls of the DAI is specified in 7.2.7.3.9.

The content access procedure determines the set of required elementary streams, requests their delivery and associates them to the scene description. The selection of a subset of elementary streams suitable for a specific ISO/IEC 14496 terminal is possible, either based on profiles or on inspection of the set of object descriptors.

7.2.7.3.2 The Initial Object Descriptor

Initial object descriptors convey information about the profiles required by the terminal compliant with ISO/IEC 14496 specifications to be able to process the described content. This profile information summarizes the complexity of the content referenced directly or indirectly through this initial object descriptor, i.e., it indicates the overall terminal capabilities required to decode and present this content. Therefore initial object descriptors constitute self-contained access points to content compliant with ISO/IEC 14496 specifications.

There are two constraints to this general statement:

- If the `includeInlineProfileLevelFlag` of the initial object descriptor is not set, the complexity of any inlined content is not included in the profile indications.
- In addition to the elementary streams that are decodable by the terminal conforming to the indicated profiles, alternate content representations might be available. This is further explained in 7.2.7.3.4.

An initial object descriptor may be conveyed by means not defined in ISO/IEC 14496. The content may be accessed starting from the elementary streams that are described by this initial object descriptor, usually one or more scene description streams and zero or more object descriptor streams.

Content refered to by an initial object descriptor may itself be referenced from another piece of ISO/IEC 14496 content. In this case, the initial object descriptor will be conveyed in an object descriptor stream and the `OD_IDs` of both initial object descriptors and ordinary object descriptors belong to the same name scope.

Ordinary object descriptors may be used as well to describe scene description and object descriptor streams. However, since they do not carry profile information, they can only be used to access content if that information is either not required by the terminal or is obtained by other means.

7.2.7.3.3 Usage of URLs in the Object Descriptor Framework

URLs in the object description framework serve to locate either inlined ISO/IEC 14496 content or the elementary stream data associated to individual audio-visual objects.

URLs in ES_Descriptors locate elementary stream data that shall be delivered as SL-packetized stream by the delivery entity associated to the current name scope. The complete description of the stream (its ES_Descriptor) is available locally.

URLs in object descriptors locate an object descriptor at a remote location. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. This implies that the description of the resources for the associated BIFS node or the inlined content is only available at the remote location. Note, however, that depending on the value of `includeInlineProfileLevelFlag` in the initial object descriptor, the global resources needed may already be known (i.e., including remote, inlined portions).

7.2.7.3.4 Selection of Elementary Streams for an Audio-Visual Object

Elementary streams are attached through their object descriptor to appropriate BIFS nodes which, in most cases, constitute the representation of a single audio-visual object in the scene. The selection of one or more ESs for each BIFS node may be governed by the profile indications that are conveyed in the initial object descriptor. All object descriptors shall at least include one elementary stream with suitable object type to satisfy the initially signaled profiles.

Additionally, object descriptors may aggregate ES_Descriptors for elementary streams that require more computing or bandwidth resources. Those elementary streams may be used by the receiving terminal if it is capable of processing them.

In case initial object descriptors do not indicate any profile and level or if profile and level indications are disregarded, an alternative to the profile driven selection of streams exists. The receiving terminal may evaluate the ES_Descriptors of all available elementary streams for each BIFS node and choose by some non-standardized way for which subset it has sufficient resources to decode them while observing the constraints specified in this Subclause.

NOTE — Some restrictions on the selection of and access to elementary streams might exist if a set of elementary streams shares a single object time base (see 7.3.2.6).

7.2.7.3.5 Content access in “push” and “pull” scenarios

In an interactive, or “pull” scenario, the receiving terminal actively requests the establishment of sessions and the delivery of content, i.e., streams. This usually involves a session and channel set up protocol between sender and receiver. This protocol is not specified here. However, the conceptual steps to be performed are the same in all cases and are specified in the subsequent Clauses.

In a broadcast, or “push” scenario, the receiving terminal passively processes what it receives. Instead of issuing requests for session or channel set up the receiving terminal shall evaluate the relevant descriptive information that associates ES_IDs to their transport channel. The syntax and semantics of this information is outside the scope of ISO/IEC 14496, however, it needs to be present in any delivery layer implementation. This allows the terminal to gain access to the elementary streams forming part of the content.

7.2.7.3.6 Content access through a known Object Descriptor

7.2.7.3.6.1 Pre-conditions

- An object descriptor has been acquired. This may be an initial object descriptor.
- The object descriptor contains ES_Descriptors pointing to object descriptor stream(s) and scene description stream(s) using ES_IDs.
- A communication session to the source of these streams is established.
- A mechanism exists to open a channel that takes user data as input and provides some returned data as output.

7.2.7.3.6.2 Content Access Procedure

The content access procedure shall be equivalent to the following:

1. The object descriptor is evaluated and the ES_ID for the streams that are to be opened are determined.
2. Requests for opening the selected ESs are made, using a suitable channel set up mechanism with the ES_IDs as parameter.
3. The channel set up mechanism shall return handles to the streams that correspond to the requested list of ESs.
4. Requests for delivery of the selected ESs are made.
5. Interactive scenarios: Delivery of streams starts. All scenarios: The streams now become accessible.
6. Scene description and object descriptor stream are evaluated.
7. Further streams are opened as needed with the same procedure, starting at step 1.

7.2.7.3.7 Content access through a URL in an Object Desciptor

7.2.7.3.7.1 Pre-conditions

- A URL to an object descriptor or an initial object descriptor has been acquired.
- A mechanism exists to open a communication session that takes a URL as input and provides some returned data as output.

7.2.7.3.7.2 Content access procedure

The content access procedure shall be equivalent to the following:

1. A connection to the source of the URL is made, using a suitable service set up call.
2. The service set up call shall return data consisting of a single object descriptor.
3. Continue at step 1 in 7.2.7.3.6.2.

7.2.7.3.8 Content access through a URL in an elementary stream descriptor

7.2.7.3.8.1 Pre-conditions

- An ES_Descriptor pointing to a stream through a URL has been aquired. (Note that the ES_Descriptor fully specifies the configuration of the stream.)
- A mechanism exists to open a communication session that takes a URL as input and provides some returned data as output.
- A mechanism exists to open a channel that takes user data as input and provides some returned data as output.

7.2.7.3.8.2 Content access procedure

The content access procedure shall be equivalent to the following:

1. A request to open the communication session is made, using a suitable session set up mechanism with the URL as parameter.

2. The session set up mechanism shall return a handle to the session that corresponds to the requested URL.

3. Request to open the stream is made, using a suitable channel set up mechanism.

4. The channel set up mechanism shall return a handle to the stream that corresponds to the originally requested URL.

5. Requests for delivery of the selected stream are made.

6. Interactive scenarios: Delivery of stream starts. All scenarios: The stream now becomes accessible.

EXAMPLE — Access to Complex Content

The example in Figure 7 shows a complex piece of ISO/IEC 14496 content, consisting of three parts. The upper part is a scene accessed through its initial object descriptor. It contains, among others a visual and an audio stream. A second part of the scene is inlined and accessed through its initial object descriptor that is pointed to (via URL) in the object descriptor stream of the first scene. Utilization of the initial object descriptor allows the signaling of profile information for the second scene. Therefore this scene may also be used without the first scene. The second scene contains, among others, a scaleably encoded visual object and an audio object. A third scene is inlined and accessed via the ES_IDs of its object descriptor and scene description streams. These ES_IDs are known from an object descriptor conveyed in the object descriptor stream of the second scene. Note that this third scene is not accessed through an initial object descriptor. Therefore the profile information for this scene need to be included in the profile information for the second scene.

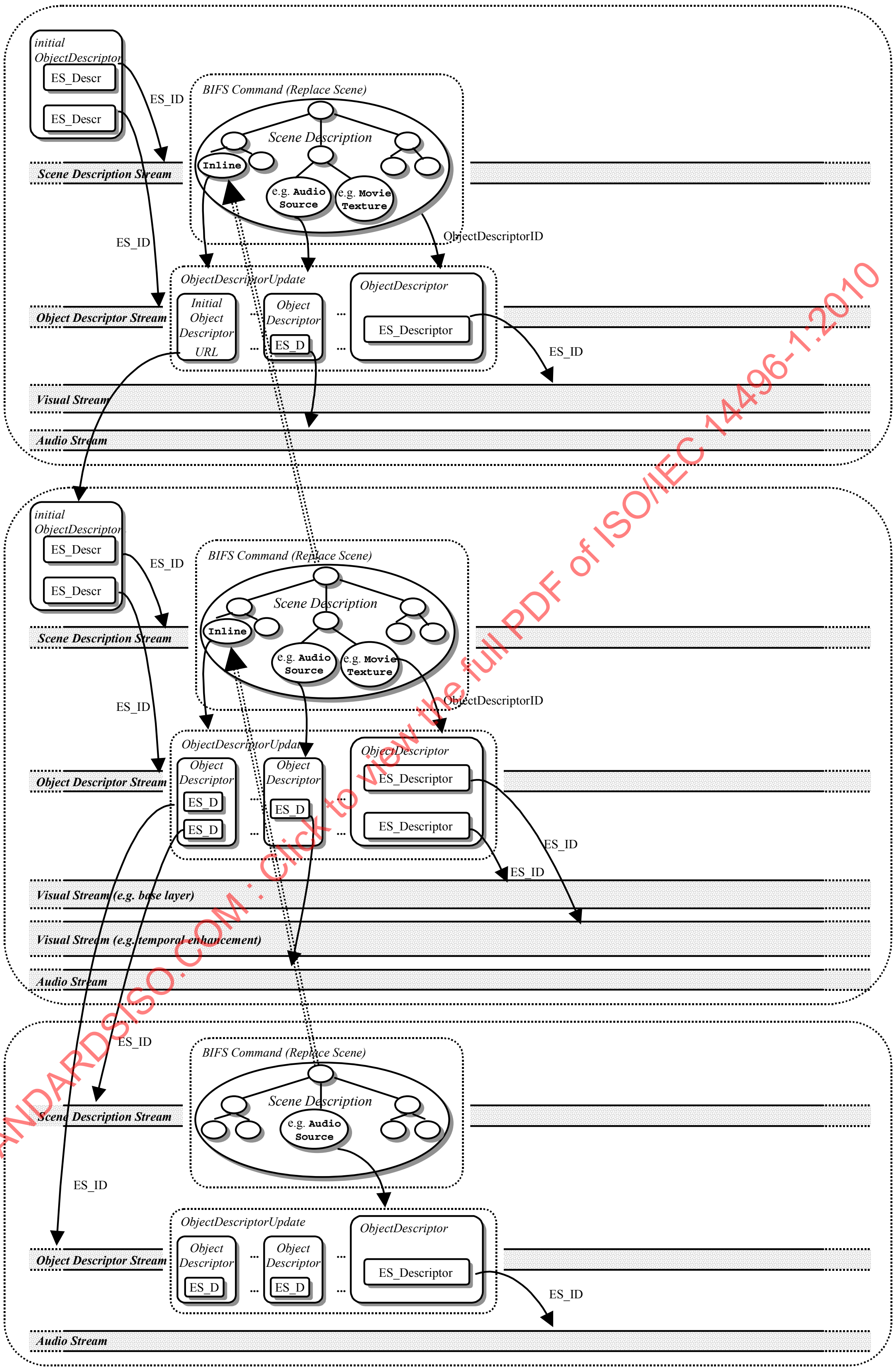


Figure 7 — Complex content example

7.2.7.3.9 Mapping of Content Access Procedure to DAI calls

The following two DAI primitives, quoted from 10.4 of ISO/IEC 14496-6, are required to implement the content access procedure described in 7.2.7.3.6 to 7.2.7.3.8:

DA_ServiceAttach (IN: URL, uuDataInBuffer, uuDataInLen;

OUT: response, serviceSessionId, uuDataOutBuffer, uuDataOutLen)

DA_ChannelAdd (IN: serviceSessionId, loop(qosDescriptor, direction, uuDataInBuffer, uuDataInLen);

OUT: loop(response, channelHandle, uuDataOutBuffer, uuDataOutLen))

DA_ServiceAttach is used to implement steps 1 and 2 of 7.2.7.3.7.2. The URL shall be passed to the IN: URL parameter. UuDataInBuffer shall remain empty. The returned serviceSessionId shall be kept for future reference to this URL. UuDataOutBuffer shall contain a single object descriptor.

DA_ChannelAdd is used to implement steps 0 and 3 of 7.2.7.3.6.2. serviceSessionId shall be the identifier for the service session that has supplied the object descriptor that includes the ES_Descriptor that is currently processed. QosDescriptor shall be the QoS_Descriptor of this ES_Descriptor, direction shall indicate upstream or downstream channels according to the `DecoderConfigDescriptor.upstream` flag. UuDataInBuffer shall contain the ES_ID of this ES_Descriptor. On successful return, channelHandle shall contain a valid, however, not normative handle to the accessible stream.

DA_ChannelAdd is used to implement steps 1 and 2 of 7.2.7.3.8.2. serviceSessionId shall be the identifier for the service session that has supplied the object descriptor that includes the ES_Descriptor that is currently processed. QosDescriptor shall be the QoS_Descriptor of this ES_Descriptor, direction shall indicate upstream or downstream channels according to the `DecoderConfigDescriptor.upstream` flag. UuDataInBuffer shall contain the URL of this ES_Descriptor. On successful return, channelHandle shall contain a valid, however, not normative handle to the accessible stream.

NOTE1 — It is a duty of the service to discriminate between the two cases with either ES_ID or URL as parameters to uuDataInBuffer in DA_ChannelAdd.

NOTE2 — Step 4 in 7.2.7.3.6.2and step 3 in 7.2.7.3.8.2 are currently not mapped to a DAI call in a normative way. It may be implemented using the DA_UserCommand() primitive.

The set up example in the following figure conveys an initial object descriptor that points to one SceneDescriptionStream, an optional ObjectDescriptorStream and additional optional SceneDescriptionStreams or ObjectDescriptorStreams. The first request to the DAI will be a DA_ServiceAttach() with the content address as a parameter. This call will return an initial object descriptor. The ES_IDs in the contained ES_Descriptors will be used as parameters to a DA_ChannelAdd() that will return handles to the corresponding channels.

Additional streams (if any) that are identified when processing the content of the object descriptor stream(s) are subsequently opened using the same procedure. The object descriptor stream is not required to be present if no further audio- or visual streams or inlined scene description streams form part of the content.

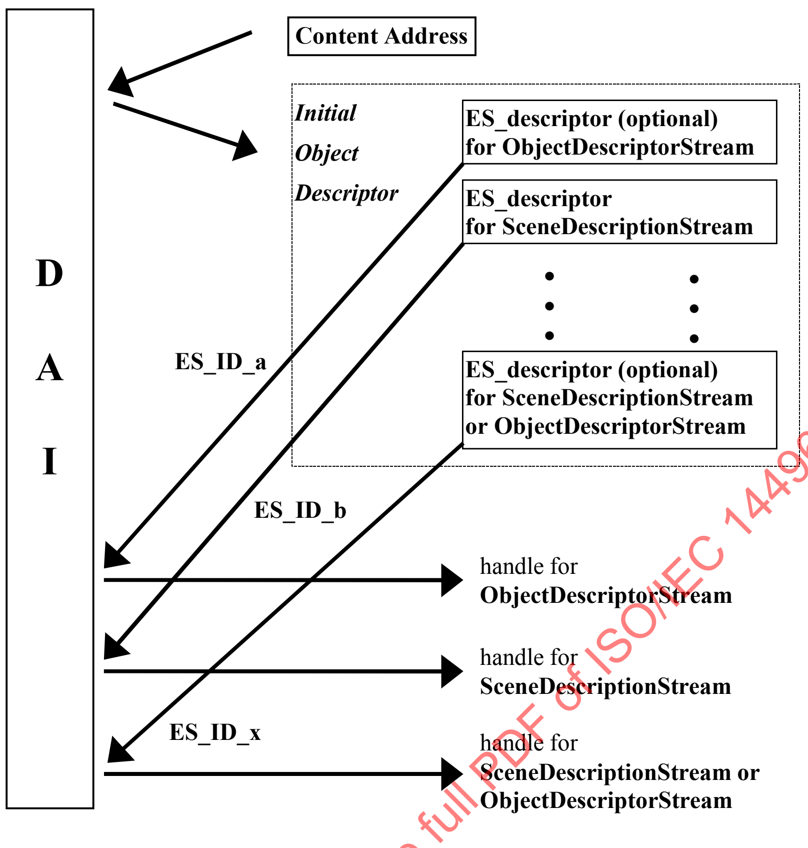

Content Address
D
A
I
Initial
Object
Descriptor
ES_descriptor (optional) for ObjectDescriptorStream
ES_descriptor for SceneDescriptionStream
ES_descriptor (optional) for SceneDescriptionStream or ObjectDescriptorStream
ES_ID_a
ES_ID_b
ES_ID_x
handle for ObjectDescriptorStream
handle for SceneDescriptionStream
handle for SceneDescriptionStream or ObjectDescriptorStream

Figure 8 — Requesting stream delivery through the DAI

7.2.8 Usage of the IPMP System interface

7.2.8.1 Overview

IPMP elementary streams and descriptors may be used in a variety of ways. For instance, IPMP elementary streams may convey time-variant IPMP information such as keys that change periodically. An IPMP elementary stream may be associated with a given elementary stream or set of elementary streams. Similarly, IPMP descriptors may be used to convey time-invariant or slowly changing IPMP information associated with a given elementary stream or set of elementary streams. This Subclause specifies methods how to associate an IPMP system to an elementary stream or a set of elementary streams. ISO/IEC 14496-13 specifies the following IPMP tools related methods:

a. Indicate IPMP Tools required for the processing of a given MPEG-4 presentation.
b. Associate an IPMP Tool to a specified Control Point of an elementary stream or set of elementary Streams.
c. Perform Mutual Authentication between IPMP Tools and between IPMP Tools and the Terminal.
d. Request the instantiation of one or more IPMP Tools by another IPMP Tool.
e. Request and receive notification of the instantiation of IPMP Tools.
f. Provide a communication channel between IPMP Tools and the User.

7.2.8.2 Association of an IPMP System with ISO/IEC 14496 content

7.2.8.2.1 Association in the initial object descriptor

An IPMP System may be associated with ISO/IEC 14496 content in the initial object descriptor. In that case the initial object descriptor shall aggregate in addition to the ES_Descriptors for scene description and object descriptor streams one or more ES_Descriptors that reference one or more IPMP elementary streams. This implies that all the elementary streams that are described through this initial object descriptor are governed by the one or more IPMP Systems that are identified within the one or more IPMP streams.

7.2.8.2.2 Association in other object descriptors

An IPMP System may be associated with ISO/IEC 14496 content in an object descriptor in three ways:

In the first case, the object descriptor aggregates in addition to the ES_Descriptors for the content elementary streams one or more ES_Descriptors that reference one or more IPMP elementary streams. This implies that all the content elementary streams described through this object descriptor are governed by the one or more IPMP Systems that are identified within the one or more IPMP streams. Note that an ES_Descriptor that describes an IPMP stream may contain references to IPMP_Descriptors.

The second method is to include one or more IPMP_DescriptorPointers in the object descriptor. This implies that all content elementary streams described by this object descriptor are governed by the IPMP System(s) that is/are identified within the referenced IPMP descriptor(s).

The third method is to include IPMP_DescriptorPointers in the ES_Descriptors embedded in this object descriptor. This implies that the elementary stream referenced by such an ES_Descriptor is controlled by an IPMP System.

7.2.8.3 IPMP of Object Descriptor streams

Object Descriptor streams shall not be affected by IPMP Systems, i.e., they shall always be available without protection by IPMP Systems. However, management may be applied using IPMP Tools.

IPMP_Descriptors, which reference one or more IPMP Tools, may be directly included in an Object Descriptor for use by elementary streams referenced within the same Object Descriptor.

The scope of the IPMP_Descriptors included and used in this way is limited to only the Object Descriptor itself and the streams defined by reference within the Object Descriptor and may not be referenced by any subsequent descriptors which may be included in the streams referenced in the Object Descriptor.

Additionally, IPMP_Tools referenced in this way shall not receive updates either by IPMP Streams or IPMP descriptor updates.

7.2.8.4 IPMP of Scene Description streams

Scene description streams are treated like any media stream, i.e. they may be managed by an IPMP System.

An IPMP_Descriptor associated with a scene description stream implies that the IPMP System controls this scene description stream.

There are two ways to protect part of a scene description (or to apply different IPMP Systems to different components of a given scene):

The first method exploits the fact that it is permissible to have more than one scene description stream associated with one object descriptor (see 7.2.7.2.2). Such a split of the scene description can be freely designed by a content author, for example, putting a basic scene description into the first stream and adding one or more additional scene description streams that enhance this basic scene using BIFS updates.

The second method is to structure the scene using one or more **Inline** nodes (see ISO/IEC 14496-11). Each **Inline** node refers to one or more additional scene description streams, each of which might use a different IPMP System.

7.2.8.5 Usage of URLs in managed and protected content

7.2.8.5.1 URLs in the BIFS Scene Description

ISO/IEC 14496 does not specify compliance points for content that uses BIFS URLs that do not point to an object descriptor. Equally, no normative way to apply an IPMP System to such links exists. The behavior of an IPMP-enabled terminal that encounters such links is undefined.

7.2.8.5.2 URLs in Object Descriptors

URLs in object descriptors point to other remote object descriptors. This merely constitutes an indirection and should not adversely affect the behavior of the IPMP System that might be invoked through this remote object descriptor.

NOTE — The only difference is that while the original site might be trusted, the referred one might not. Further corrective actions to guard against this condition are not in the scope of ISO/IEC 14496.

7.2.8.5.3 URLs in ES_Descriptors

URLs in ES descriptors are used to access elementary streams remotely. This merely constitutes an indirection and therefore does not adversely affect the behavior of the IPMP System that might be invoked through this remote object descriptor.

NOTE — The only difference is that while the original site might be trusted, the referred one might not. Further corrective actions to guard against this condition are not in the scope of ISO/IEC 14496.

7.2.8.6 IPMP Decoding Process

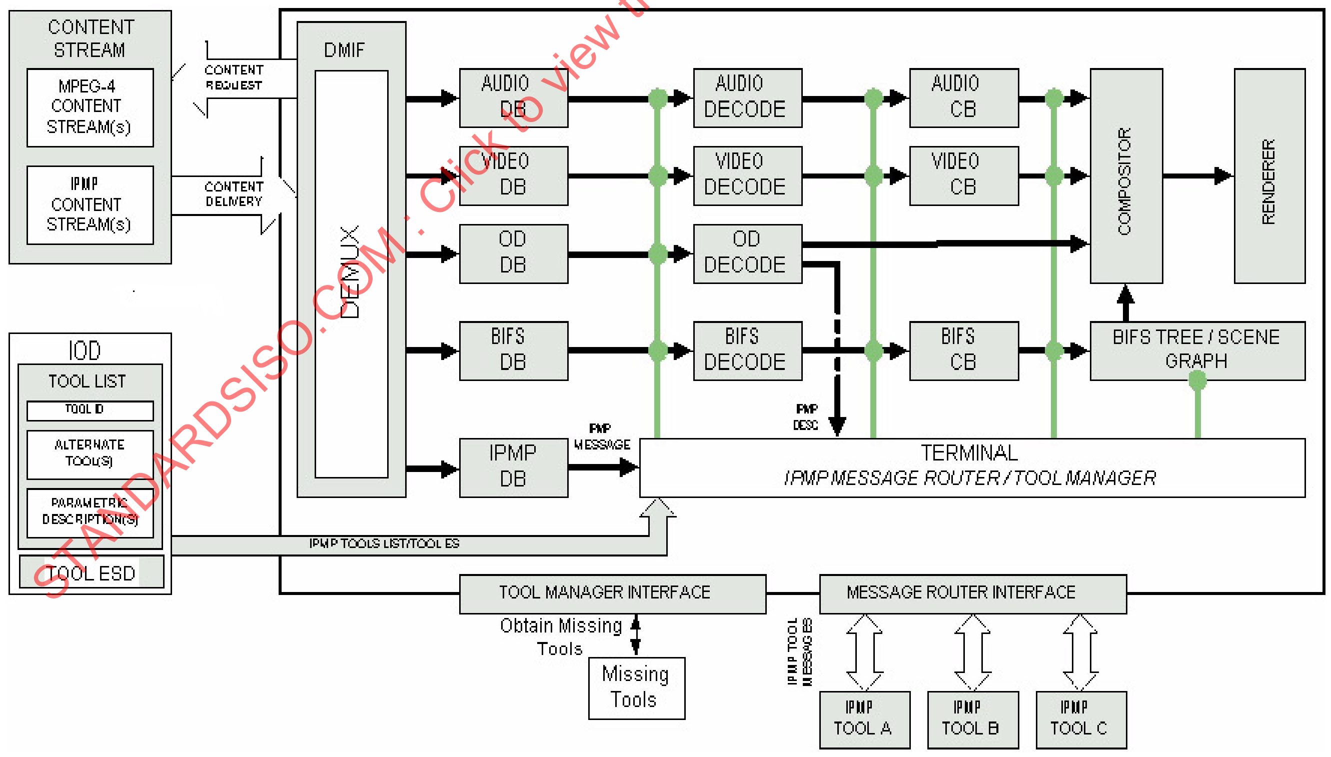


Figure 9 — IPMP system in the ISO/IEC 14496 terminal architecture

Figure 9 depicts the injection of IPMP systems or tools with respect to the MPEG-4 terminal. IPMP specific data is supplied to the IPMP systems or tools via IPMP streams and/or IPMP descriptors, and the IPMP systems or tools releases protected content *after the sync layer.*

Each elementary stream under the control of IPMP systems or tools has the conceptual element of a *stream flow controller.* Stream flow control can take place between the the SyncLayer decoder and the decoder buffer. As the figure indicates, elements of IPMP control may take place at other points in the terminal including, after decoding (as with some watermarking systems) or in the decoded BIFS stream, or after the composition buffers have been written, or in the BIFS scene tree. Stream flow controllers either enable or disable processing of an elementary stream in a non-normative way that depends on the status information provided by the IPMP systems or tools.

Finally, the IPMP systems or tools must at a minimum:

1. Process the IPMP stream and descriptor

2. Appropriately manage (e.g. decrypt and release) protected elementary streams.

The initialization process of the IPMP systems or tools is not specified except that it shall not unduly delay the content access process as specified in 7.2.7.3.

7.3 Synchronization of Elementary Streams

7.3.1 Introduction

This Subclause defines the tools to maintain temporal synchronisation within and among elementary streams. The conceptual elements that are required for this purpose, namely time stamps and clock reference information, have already been introduced in 7.1. The syntax and semantics to convey these elements to a receiving terminal are embodied in the sync layer, specified in 7.3.2. This syntax is configurable to adapt to the needs of different types of elementary streams. The required configuration information is specified in 7.3.2.3.

On the sync layer, an elementary stream is mapped into a sequence of packets, called an SL-packetized stream (SPS). Packetization information has to be exchanged between the entity that generates an elementary stream and the sync layer. This relation may be described by a conceptual elementary stream interface (ESI) between both layers (see Annex G). The ESI is a concept to explain the information flow between layers, however, need not be accessible in an implementation.

SL-packetized streams are conveyed through a delivery mechanism that is outside the scope of ISO/IEC 14496-1. This delivery mechanism is only described in terms of the DMIF Application Interface (DAI) whose semantics are specified in ISO/IEC 14496-6. It specifies the information that needs to be exchanged between the sync layer and the delivery mechanism. The basic data transport feature that this delivery mechanism shall provide is the framing of the data packets generated by the sync layer. The DAI is a reference point that need not be accessible in an implementation. The required properties of the DAI are described in 7.3.3.

The items specified in this Clause are depicted in Figure 10 below.

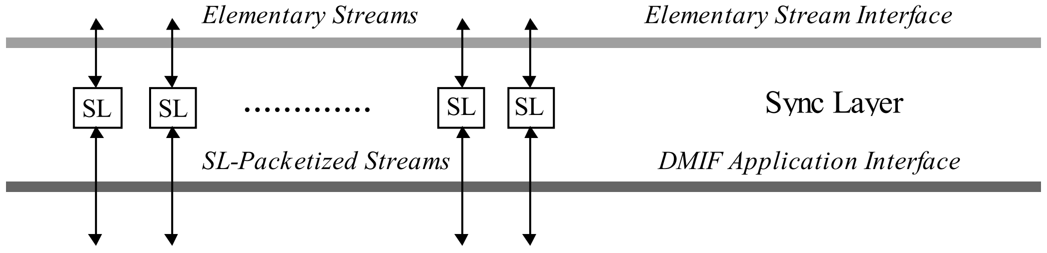


Figure 10 — The Sync Layer

7.3.2 Sync Layer

7.3.2.1 Overview

The sync layer (SL) specifies a syntax for the packetization of elementary streams into access units or parts thereof. Such a packet is called SL packet. The sequence of SL packets resulting from one elementary stream is called an SL-packetized stream (SPS). Access units are the only semantic entities at this layer that need to be preserved from end to end. Their content is opaque. Access units are used as the basic unit for synchronisation.

An SL packet consists of an SL packet header and an SL packet payload. The SL packet header provides means for continuity checking in case of data loss and carries the coded representation of the time stamps and associated information. The detailed semantics of the time stamps are specified in 7.1.3 that defines the timing aspects of the systems decoder model. The SL packet header is configurable as specified in 7.3.2.3. The SL packet header itself is specified in 7.3.2.4.

An SL packet does not contain an indication of its length. Therefore, SL packets must be framed by a suitable lower layer protocol using, e.g., the M4Mux tool specified in 7.4. Consequently, an SL-packetized stream is not a self-contained data stream that can be stored or decoded without such framing.

An SL-packetized stream does not provide identification of the ES_ID associated to the elementary stream (see 7.2.6.5) in the SL packet header. This association must be conveyed through a stream map table using the appropriate signalling means of the delivery mechanism.

7.3.2.2 SL Packet Specification

7.3.2.2.1 Syntax

```
class SL_Packet (SLConfigDescriptor SL) {
  aligned(8) SL_PacketHeader slPacketHeader(SL);
  aligned(8) SL_PacketPayload slPacketPayload;
}
```

7.3.2.2.2 Semantics

In order to properly parse an `SL_Packet`, it is required that the `SLConfigDescriptor` for the elementary stream to which the SL_Packet belongs is known, since the `SLConfigDescriptor` conveys the configuration of the syntax of the SL packet header.

`slPacketHeader` – an `SL_PacketHeader` element as specified in 7.3.2.4.

`slPacketPayload` – an `SL_PacketPayload` that contains an opaque payload.

7.3.2.3 SL Packet Header Configuration

7.3.2.3.1 Syntax

```
class SLConfigDescriptor extends BaseDescriptor : bit(8) tag=SLConfigDescrTag {
   bit(8) predefined;
   if (predefined==0) {
      bit(1) useAccessUnitStartFlag;
      bit(1) useAccessUnitEndFlag;
      bit(1) useRandomAccessPointFlag;
      bit(1) hasRandomAccessUnitsOnlyFlag;
      bit(1) usePaddingFlag;
      bit(1) useTimeStampsFlag;
      bit(1) useIdleFlag;
      bit(1) durationFlag;
      bit(32) timeStampResolution;
      bit(32) OCRResolution;
      bit(8) timeStampLength; // must be ≤ 64
      bit(8) OCRLength;     // must be ≤ 64
      bit(8) AU_Length;     // must be ≤ 32
      bit(8) instantBitrateLength;
      bit(4) degradationPriorityLength;
      bit(5) AU_seqNumLength; // must be ≤ 16
      bit(5) packetSeqNumLength; // must be ≤ 16
      bit(2) reserved=0b11;
   }
   if (durationFlag) {
      bit(32) timeScale;
      bit(16) accessUnitDuration;
      bit(16) compositionUnitDuration;
   }
   if (!useTimeStampsFlag) {
      bit(timeStampLength) startDecodingTimeStamp;
      bit(timeStampLength) startCompositionTimeStamp;
   }
}

class ExtendedSLConfigDescriptor extends SLConfigDescriptor : bit(8)
tag=ExtSLConfigDescrTag {
   SLExtensionDescriptor slextDescr[1..255];
}
```

7.3.2.3.2 Semantics

The SL packet header may be configured according to the needs of each individual elementary stream. Parameters that can be selected include the presence, resolution and accuracy of time stamps and clock references. This flexibility allows, for example, a low bitrate elementary stream to incur very little overhead on SL packet headers.

For each elementary stream the configuration is conveyed in an `SLConfigDescriptor`, which is part of the associated `ES_Descriptor` within an object descriptor.

The configurable parameters in the SL packet header can be divided in two classes: those that apply to each SL packet (e.g. OCR, sequenceNumber) and those that are strictly related to access units (e.g. time stamps, accessUnitLength, instantBitrate, degradationPriority).

`predefined` – allows to default the values from a set of predefined parameter sets as detailed below.

NOTE — This table will be updated by amendments to ISO/IEC 14496 to include predefined configurations as required by future profiles.

Table 13 — Overview of predefined `SLConfigDescriptor` values

`Predefined` field value	Description
0x00	Custom
0x01	null SL packet header
0x02	Reserved for use in MP4 files
0x03 – 0xFF	Reserved for ISO use

Table 14 — Detailed predefined SLConfigDescriptor values

`Predefined` field value	0x01	0x02
`UseAccessUnitStartFlag`	0	0
`UseAccessUnitEndFlag`	0	0
`UseRandomAccessPointFlag`	0	0
`UsePaddingFlag`	0	0
`UseTimeStampsFlag`	0	1
`UseIdleFlag`	0	0
`DurationFlag`	0	0
`TimeStampResolution`	1000	-
`OCRResolution`	-	-
`TimeStampLength`	32	0
`OCRlength`	-	0
`AU_length`	0	0
`InstantBitrateLength`	-	0
`DegradationPriorityLength`	0	0
`AU_seqNumLength`	0	0
`PacketSeqNumLength`	0	0

`useAccessUnitStartFlag` – indicates that the `accessUnitStartFlag` is present in each SL packet header of this elementary stream.

`useAccessUnitEndFlag` – indicates that the `accessUnitEndFlag` is present in each SL packet header of this elementary stream.

If neither `useAccessUnitStartFlag` nor `useAccessUnitEndFlag` are set this implies that each SL packet corresponds to a complete access unit.

`useRandomAccessPointFlag` – indicates that the `RandomAccessPointFlag` is present in each SL packet header of this elementary stream.

`hasRandomAccessUnitsOnlyFlag` – indicates that each SL packet corresponds to a random access point. In that case the `randomAccessPointFlag` need not be used.

`usePaddingFlag` – indicates that the `paddingFlag` is present in each SL packet header of this elementary stream.

UseTimeStampsFlag: indicates that time stamps are used for synchronisation of this elementary stream. They are conveyed in the SL packet headers. Otherwise, the parameters accessUnitDuration, compositionUnitDuration, startDecodingTimeStamp and startCompositionTime-Stamp conveyed in this SL packet header configuration shall be used for synchronisation.

NOTE — The use of start time stamps and durations (useTimeStampsFlag=0) may only be feasible under some conditions, including an error free environment. Random access is not easily possible.

useIdleFlag – indicates that idleFlag is used in this elementary stream.

durationFlag – indicates that the constant duration of access units and composition units for this elementary stream is subsequently signaled.

timeStampResolution – is the resolution of the time stamps in clock ticks per second.

OCRResolution – is the resolution of the object time base in cycles per second.

timeStampLength – is the length of the time stamp fields in SL packet headers. timeStampLength shall take values between zero and 64 bit.

OCRlength – is the length of the objectClockReference field in SL packet headers. A length of zero indicates that no objectClockReferences are present in this elementary stream. If OCRstreamFlag is set, OCRLength shall be zero. Else OCRlength shall take values between zero and 64 bit.

AU_Length – is the length of the accessUnitLength fields in SL packet headers for this elementary stream. AU_Length shall take values between zero and 32 bit.

instantBitrateLength – is the length of the instantBitrate field in SL packet headers for this elementary stream.

degradationPriorityLength – is the length of the degradationPriority field in SL packet headers for this elementary stream.

AU_seqNumLength – is the length of the AU_sequenceNumber field in SL packet headers for this elementary stream.

packetSeqNumLength – is the length of the packetSequenceNumber field in SL packet headers for this elementary stream.

timeScale – used to express the duration of access units and composition units. One second is evenly divided in timeScale parts.

accessUnitDuration – the duration of an access unit is accessUnitDuration * 1/timeScale seconds.

compositionUnitDuration – the duration of a composition unit is compositionUnitDuration * 1/timeScale seconds.

startDecodingTimeStamp – conveys the time at which the first access unit of this elementary stream shall be decoded. It is conveyed in the resolution specified by timeStampResolution.

startCompositionTimeStamp – conveys the time at which the composition unit corresponding to the first access unit of this elementary stream shall be decoded. It is conveyed in the resolution specified by timeStampResolution.

slextDescr – is an array of ExtensionDescriptors defined for ExtendedSLConfigDescriptor as specified in 7.3.2.3.1.

7.3.2.3.3 SLExtentionDescriptor Syntax

```
abstract class SLExtensionDescriptor : bit(8) tag=0 {
}

class DependencyPointer extends SLExtensionDescriptor: bit(8) tag=
DependencyPointerTag {
   bit(6) reserved;
   bit(1) mode;
   bit(1) hasESID;
   bit(8) dependencyLength;
   if (hasESID)
   {
      bit(16) ESID;
   }
}
class MarkerDescriptor extends SLExtensionDescriptor: bit(8)
tag=DependencyMarkerTag {
   int(8) markerLength;
}
```

7.3.2.3.4 SLExtentionDescriptor Semantics

SLExtensionDescriptor is an abstract class specified so as to be the base class of sl extensions.

7.3.2.3.4.1 DependencyPointer Semantics

DependencyPointer extends SLExtensionDescriptor and specifies that access units from this stream depend on another stream.

If mode equals 0, the latter stream can be identified through the ESID field or if no ESID is present, using the `dependsOn_ES_ID` ESID, and access units from this stream will point to the decodingTimeStamps of that stream.

If mode equals 1, access units from this stream will convey identifiers, for which the system (e.g. IPMP tools) are responsible to know whether dependent resources (e.g. keys) are available.

In both cases, the length of this pointer or identifier is `dependencyLength`.

If mode is 0 then `dependencyLength` shall be the length of the `decodingTimeStamp`.

7.3.2.3.4.2 MarkerDescriptor Semantics

MarkerDescriptor extends SLExtensionDescriptor and allows to tag access units so as to be able to refer to them independently from their decodingTimeStamp.

`markerLength` – is the length for identifiers tagging access units.

7.3.2.4 SL Packet Header Specification

7.3.2.4.1 Syntax

```
aligned(8) class SL_PacketHeader (SLConfigDescriptor SL) {
   if (SL.useAccessUnitStartFlag)
      bit(1) accessUnitStartFlag;
   if (SL.useAccessUnitEndFlag)
      bit(1) accessUnitEndFlag;
   if (SL.OCRLength>0)
```

```
      bit(1) OCRflag;
   if (SL.useIdleFlag)
      bit(1) idleFlag;
   if (SL.usePaddingFlag)
      bit(1) paddingFlag;
   if (paddingFlag)
      bit(3) paddingBits;

   if (!idleFlag && (!paddingFlag || paddingBits!=0)) {
      if (SL.packetSeqNumLength>0)
         bit(SL.packetSeqNumLength) packetSequenceNumber;
      if (SL.degradationPriorityLength>0)
         bit(1) DegPrioflag;
      if (DegPrioflag)
         bit(SL.degradationPriorityLength) degradationPriority;
      if (OCRflag)
         bit(SL.OCRLength) objectClockReference;

      if (accessUnitStartFlag) {
         if (SL.useRandomAccessPointFlag)
            bit(1) randomAccessPointFlag;
         if (SL.AU_seqNumLength >0)
            bit(SL.AU_seqNumLength) AU_sequenceNumber;
         if (SL.useTimeStampsFlag) {
            bit(1) decodingTimeStampFlag;
            bit(1) compositionTimeStampFlag;
         }
         if (SL.instantBitrateLength>0)
            bit(1) instantBitrateFlag;
         if (decodingTimeStampFlag)
            bit(SL.timeStampLength) decodingTimeStamp;
         if (compositionTimeStampFlag)
            bit(SL.timeStampLength) compositionTimeStamp;
         if (SL.AU_Length > 0)
            bit(SL.AU_Length)     accessUnitLength;
         if (instantBitrateFlag)
            bit(SL.instantBitrateLength)   instantBitrate;
      }
   if (SL.tag == ExtSLConfigDescrTag)
   {
      for (int i=0; i<SL.slextDescr.length;i++)
      {
         switch(SL.slextDescr[i].tag)
         {
            case DependencyPointerTag:
               Marker(SL.slextDescr[i].dependencyLength) value;
               break;
            case DependencyMarkerTag:
               Marker(SL.slextDescr[i].markerLength) value;
               break;
            default:
               break;
         }
      }
   }
   }
}

aligned expandable class Marker(int length) {
   bit(length) value;
}
```

7.3.2.4.2 Semantics

`accessUnitStartFlag` – when set to one indicates that the first byte of the payload of this SL packet is the start of an access unit. If this syntax element is omitted from the SL packet header configuration its default value is known from the previous SL packet with the following rule:

`accessUnitStartFlag` = (previous-SL packet has `accessUnitEndFlag`==1) ? 1 : 0.

`accessUnitEndFlag` – when set to one indicates that the last byte of the SL packet payload is the last byte of the current access unit. If this syntax element is omitted from the SL packet header configuration its default value is only known after reception of the subsequent SL packet with the following rule:

`accessUnitEndFlag` = (subsequent-SL packet has `accessUnitStartFlag`==1) ? 1 : 0.

If neither `AccessUnitStartFlag` nor `AccessUnitEndFlag` are configured into the SL packet header this implies that each SL packet corresponds to a single access unit, hence both `accessUnitStartFlag = accessUnitEndFlag = 1`.

NOTE — When the SL packet header is configured to use `accessUnitStartFlag` but neither `accessUnitEndFlag` nore `accessUnitLength`, it is not guaranteed that the terminal can determine the end of an access unit before the subsequent one is received.

`OCRflag` – when set to one indicates that an `objectClockReference` will follow. The default value for `OCRflag` is zero.

`idleFlag` – indicates that this elementary stream will be idle (i.e., not produce data) for an undetermined period of time. This flag may be used by the decoder to discriminate between deliberate and erroneous absence of subsequent SL packets.

`paddingFlag` – indicates the presence of padding in this SL packet. The default value for `paddingFlag` is zero.

`paddingBits` – indicate the mode of padding to be used in this SL packet. The default value for `paddingBits` is zero.

If `paddingFlag` is set and `paddingBits` is zero, this indicates that the subsequent payload of this SL packet consists of padding bytes only. `accessUnitStartFlag`, `randomAccessPointFlag` and `OCRflag` shall not be set if `paddingFlag` is set and paddingBits is zero.

If `paddingFlag` is set and `paddingBits` is greater than zero, this indicates that the payload of this SL packet is followed by `paddingBits` of zero stuffing bits for byte alignment of the payload.

`packetSequenceNumber` – if present, it shall be continuously incremented for each SL packet as a modulo counter. A discontinuity at the decoder corresponds to one or more missing SL packets. In that case, an error shall be signalled to the sync layer user. If this syntax element is omitted from the SL packet header configuration, continuity checking by the sync layer cannot be performed for this elementary stream.

Duplication of SL packets: elementary streams that have a `sequenceNumber` field in their SL packet headers may use duplication of SL packets for error resilience. The duplicated SL packet(s) shall immediately follow the original. The packet`SequenceNumber` of duplicated SL packets shall have the same value and each byte of the original SL packet shall be duplicated, with the exception of an `objectClockReference` field, if present, which shall encode a valid value for the duplicated SL packet.

`degPrioFlag` - when set to one indicates that `degradationPriority` is present in this packet.

degradationPriority – indicates the importance of the payload of this SL packet. The streamPriority defines the base priority of an ES. degradationPriority defines a decrease in priority for this SL packet relative to the base priority. The priority for this SL packet is given by:

SL_PacketPriority = streamPriority – degradationPriority

degradationPriority remains at this value until its next occurrence. This indication may be for graceful degradation by the decoder of this elementary stream as well as by the adaptor to a specific delivery layer instance. The relative amount of complexity degradation among SL packets of different elementary streams increases as SL_PacketPriority decreases.

objectClockReference – contains an Object Clock Reference time stamp. The OTB time value t is reconstructed from this OCR time stamp according to the following formula:

$$t = (\texttt{objectClockReference}/\texttt{SL.OCRResolution}) + k*(2^{\texttt{SL.OCRLength}}/\texttt{SL.OCRResolution})$$

where k is the number of times that the objectClockReference counter has wrapped around.

objectClockReference is only present in the SL packet header if OCRflag is set.

NOTE — It is possible to convey just an OCR value and no payload within an SL packet.

The following is the semantics of the syntax elements that are only present at the start of an access unit when explicitly signaled by accessUnitStartFlag in the bitstream:

randomAccessPointFlag – when set to one indicates that random access to the content of this elementary stream is possible here. randomAccessPointFlag shall only be set if accessUnitStartFlag is set. If this syntax element is omitted from the SL packet header configuration, its default value is the value of SLConfigDescriptor.hasRandomAccessUnitsOnlyFlag for this elementary stream.

AU_sequenceNumber – if present, successive access units shall either have the same sequence number or the value be continuously incremented as a modulo counter. A discontinuity at the decoder corresponds to one or more missing access units. In that case, an error shall be signaled to the sync layer user.

Duplication of access units: Access units sent using the same sequence number as the immediately preceding AU shall be ignored if and only if the second access unit is a random access point. Such a repeated access unit, where the first did not have RAP set but the repeated one does, allows random access points to be added to a broadcast stream, permitting clients to enter the stream at defined points during its transmission, whilst not disrupting clients already receiving the stream. On the other hand, reception of two access units with the same sequence number, when the second is not a RAP, means that the two access units refer to the same key state of the scene. I.e. the second access unit can be safely processed by the decoder even if it is known to the decoder that one or more access units that originally existed between the two were lost on the network.

decodingTimeStampFlag – indicates that a decoding time stamp is present in this packet.

compositionTimeStampFlag – indicates that a composition time stamp is present in this packet.

accessUnitLengthFlag – indicates that the length of this access unit is present in this packet.

instantBitrateFlag – indicates that an instantBitrate is present in this packet.

decodingTimeStamp – is a decoding time stamp as configured in the associated SLConfigDescriptor. The decoding time td of this access unit is reconstructed from this decoding time stamp according to the formula:

td = (decodingTimeStamp/SL.timeStampResolution + k * $2^{\mathrm{SL.timeStampLength}}$/SL.timeStampResolution

where k is the number of times that the decodingTimeStamp counter has wrapped around.

A decodingTimeStamp shall only be present if the decoding time is different from the composition time for this access unit.

compositionTimeStamp – is a composition time stamp as configured in the associated SLConfigDescriptor. The composition time tc of the first composition unit resulting from this access unit is reconstructed from this composition time stamp according to the formula:

td = (compositionTimeStamp/SL.timeStampResolution + k * $2^{\mathrm{SL.timeStampLength}}$/SL.timeStampResolution

where k is the number of times that the compositionTimeStamp counter has wrapped around.

accessUnitLength – is the length of the access unit in bytes. If this syntax element is not present or has the value zero, the length of the access unit is unknown.

instantBitrate – is the instantaneous bit rate in bits per second of this elementary stream until the next instantBitrate field is found.

If the SLConfigDescriptor is an ExtendedSLConfigDescriptor (i.e. its tag is ExtSLConfigDescrTag), then descriptors associated with the array of SLExtensionDescriptors are appended to the end of the SLPacket Header.

Note – Since those descriptors conveying the extended SL information; carry their size, they can be skipped by a decoder.

DependencyPointerDescriptor and MarkerDescriptor define their associated descriptors as follows :

For DependencyPointerDescriptor a Marker of length dependencyLength will be encoded. It shall resolve either to an identifier or to a decodingTimeStamp as specified in 7.3.2.3.4.1.

For MarkerDescriptor a marker of length markerLength is encoded.

7.3.2.5 Clock Reference Stream

An elementary stream of streamType = ClockReferenceStream may be declared by means of the object descriptor. It is used for the sole purpose of conveying Object Clock Reference time stamps. Multiple elementary streams in a name scope may make reference to such a ClockReferenceStream by means of the OCR_ES_ID syntax element in the SLConfigDescriptor to avoid redundant transmission of Clock Reference information. Note, however, that circular references between elementary streams using OCR_ES_ID are not permitted.

On the sync layer a ClockReferenceStream is realized by configuring the SL packet header syntax for this SL-packetized stream such that only OCR values of the required OCRresolution and OCRlength are present in the SL packet header.

There shall not be any SL packet payload present in an SL-packetized stream of streamType = ClockReferenceStream.

In the DecoderConfigDescriptor for a clock reference stream ObjectTypeIndication shall be set to '0xFF', hasRandomAccessUnitsOnlyFlag to one and bufferSizeDB to '0'.

The following indicates recommended values for the SLConfigDescriptor of a Clock Reference Stream:

Table 15 — SLConfigDescriptor parameter values for a ClockReferenceStream

useAccessUnitStartFlag	0
useAccessUnitEndFlag	0
useRandomAccessPointFlag	0
usePaddingFlag	0
useTimeStampsFlag	0
useIdleFlag	0
durationFlag	0
timeStampResolution	0
timeStampLength	0
AU_length	0
degradationPriorityLength	0
AU_seqNumLength	0

7.3.2.6 Restrictions for elementary streams sharing the same object time base

While it is possible to share an object time base between multiple elementary streams through OCR_ES_ID, a number of restrictions for the access to and processing of these elementary streams exist as follows:

1. When several elementary streams share a single object time base, the elementary streams without embedded object clock reference information shall not be used by the player, even if accessible, until the elementary stream carrying the object clock reference information becomes accessible (see 7.2.7.3 for the stream access procedure).

2. If an elementary stream without embedded object clock reference information is made available to the terminal at a later point in time than the elementary stream carrying the object clock reference information, it shall be delivered in synchronization with the other stream(s). Note that this implies that such a stream might not start playing from its beginning, depending on the current value of the object time base.

3. When an elementary stream carrying object clock reference information becomes unavailable or is otherwise manipulated in its delivery (e.g., paused), all other elementary streams which use the same object time base shall follow this behavior, i.e., become unavailable or be manipulated in the same way.

4. When an elementary stream without embedded object clock reference information becomes unavailable this has no influence on the other elementary streams that share the same object time base.

7.3.2.7 Usage of configuration options for object clock reference and time stamp values

7.3.2.7.1 Resolution of ambiguity in object time base recovery

Due to the limited length of objectClockReference values these time stamps may be ambiguous. The OTB time value can be reconstructed each time an objectClockReference is transmitted in the headers of an SL packet according to the following formula:

$t_{OTB\_reconstructed}=(\texttt{objectClockReference}/\texttt{SL.OCRResolution})+k*(2^{\texttt{SL.OCRLength}}/\texttt{SL.OCRResolution})$

with k being an integer value denoting the number of wrap-arounds. The resulting time base $t_{OTB\_reconstructed}$ is measured in seconds.

When the first `objectClockReference` for an elementary stream is acquired, the value k shall be set to one. For each subsequent occurence of `objectClockReference` the value k is estimated as follows:

The terminal shall implement a mechanism to estimate the value of the object time base for any time instant.

Each time an `objectClockReference` is received, the current estimated value of the OTB $t_{OTB\_estimated}$ shall be sampled. Then, $t_{OTB\_rec}(k)$ is evaluated for different values of k. The value k that minimizes the term $| t_{OTB\_estimated} - t_{OTB\_rec}(k)|$ shall be assumed to yield the correct value of $t_{OTB\_reconstructed}$. This value may be used as new input to the object time base estimation mechanism.

The application shall ensure that this procedure yields an unambiguous value of k by selecting an appropriate length and resolution of the `objectClockReference` element and a sufficiently high frequency of insertion of `objectClockReference` values in the elementary stream. The choices for these values depend on the delivery jitter for SL packets as well as the anticipated maximum drift between the clocks of the transmitting and receiving terminal.

7.3.2.7.2 Resolution of ambiguity in time stamp recovery

Due to the limited length of `decodingTimeStamp` and `compositionTimeStamp` values these time stamps may become ambiguous according to the following formula:

$t_{ts}(m)=(\texttt{TimeStamp/SL.timeStampResolution})+m*(2^{\texttt{SL.timeStampLength}}/\texttt{SL.timeStampResolution})$

with `TimeStamp` being either a `decodingTimeStamp` or a `compositionTimeStamp` and m being an integer value denoting the number of wrap-arounds.

The correct value $t_{timestamp}$ of the time stamp can be estimated as follows:

Each time a `TimeStamp` is received, the current estimated value of the OTB $t_{OTB\_estimated}$ shall be sampled. $t_{ts}(m)$ is evaluated for different values of m. The value m that minimizes the term $| t_{OTB\_estimated} – t_{ts}(m)|$ shall be assumed to yield the correct value of $t_{timestamp}$.

The application may choose, separately for every individual elementary stream, the length and resolution of time stamps so as to match its requirements on unambiguous positioning of time events. This choice depends on the maximum time that an SL packet with a `TimeStamp` may be sent prior to the point in time indicated by the `TimeStamp` as well as the required precision of temporal positioning.

7.3.2.7.3 Usage considerations for object clock references and time stamps

The time line of an object time base allows to discriminate two time instants separated by more than 1/`SL.OCRResolution`. `OCRResolution` should be chosen sufficiently high to match the accuracy needed by the application to synchronize a set of elementary streams.

The decoding and composition time stamp allow to discriminate two time instants separated by more than 1/`SL.timeStampResolution`. `timeStampResolution` should be chosen sufficiently high to match the accuracy needed by the application in terms of positioning of access units for a given elementary stream.

A `TimeStampResolution` higher than the `OCRResolution` will not achieve better discrimination between events. If `TimeStampResolution` is lower than the `OCRResolution`, events for this specific stream cannot be positioned with the maximum precision possible with this given `OCRResolution`.

The parameter `OCRLength` is signaled in the SL header configuration. $2^{\texttt{SL.OCRLength}}/\texttt{SL.OCRResolution}$ is the time interval covered by the `objectClockReference` counter before it wraps around. `OCRLength` should be chosen sufficiently high to match the application needs for unambiguous positioning of time events from a set of elementary streams.

When an application knows the value k defined in 7.3.2.7.1, the OTB time line is unambiguous for any time value. When the application cannot reconstruct the k factor, as for example in any application that permits random access without additional side information, the OTB time line is ambiguous modulo $2^{\mathtt{SL.OCRLength}}$/`SL.OCRResolution`. Therefore, any time stamp refering to this OTB is ambiguous. Therefore, any time stamp refering to this OTB is ambiguous. It may, however, be considered unambiguous within an application environment through knowledge about the maximum expected delivery jitter and constraints on the time by which an access unit can be sent prior to its decoding time.

Note that elementary streams that choose the time interval $2^{\mathtt{SL.timeStampLength}}$/`SL.timeStampResolution` higher than $2^{\mathtt{SL.OCRLength}}$/`SL.OCRResolution` can still only position time events unambiguously in the smaller of the two intervals.

In cases, where k and m can not be estimated correctly, the buffer model may be violated, resulting in unpredictable performance of the decoder.

EXAMPLE — Let's assume an application that wants to synchronize elementary streams with a precision of 1 ms. `OCRResolution` should be chosen equal to or higher than 1000 (the time between two successive ticks of the OCR is then equal to 1ms). Let's assume `OCRResolution=2000`.

The application assumes a drift between the STB and the OTB of 0.1% (i.e. 1ms every second). The clocks need therefore to be adjusted at least every second (i.e. in the worst case, the clocks will have drifted 1ms which is the precision constraint). Let's assume that `objectClockReference` are sent every 1s.

The application wants to have an unambiguous OTB time line of 24h without need to reconstruct the k factor. The `OCRLength` is therefore chosen accordingly such that $2^{\mathtt{SL.OCRLength}}$/`SL.OCRResolution=24h`.

Let's assume now that the application wants to synchronize events within a single elementary stream with a precision of 10 ms. `TimeStampResolution` should be chosen equal to or higher than 100 (the time between two successive ticks of the `TimeStamp` is then equal to 10ms). Let's assume `TimeStampResolution=200`.

The application wants to be able to send access units at maximum 1 minute ahead of their decoding or composition time. The `timeStampLength` is therefore chosen as

$2^{\mathtt{SL.timeStampLength}}$/`SL.timeStampResolution = 2 minutes`.

7.3.3 DMIF Application Interface

The DMIF Application Interface is a conceptual interface that specifies which data need to be exchanged between the sync layer and the delivery mechanism. Communication between the sync layer and the delivery mechanism includes SL-packetized data as well as additional information to convey the length of each SL packet.

An implementation of ISO/IEC 14496-1 does not have to expose the DMIF Application Interface. A terminal compliant with ISO/IEC 14496-1, however, shall have the functionality described by the DAI to be able to receive the SL packets that constitute an SL-packetized stream. Specifically, the delivery mechanism below the sync layer shall supply a method to frame or otherwise encode the length of the SL packets transported through it.

The DMIF Application Interface specified in ISO/IEC 14496-6 embodies a superset of the required data delivery functionality. The DAI has data primitives to receive and send data, which include indication of the data size. With this interface, each invocation of a DA_Data or a DA_DataCallback shall transfer one SL packet between the sync layer and the delivery mechanism below.

7.4 Multiplexing of Elementary Streams

7.4.1 Introduction

Elementary stream data encapsulated in SL-packetized streams are sent/received through the DMIF Application Interface, as specified in 7.3. Multiplexing procedures and the architecture of the delivery protocol layers are outside the scope of ISO/IEC 14496-1. However, care has been taken to define the sync layer syntax and semantics such that SL-packetized streams can be easily embedded in various transport protocol stacks.

The analysis of existing transport protocol stacks has shown that, for stacks with fixed length packets (e.g., MPEG-2 Transport Stream) or with high multiplexing overhead (e.g., RTP/UDP/IP), it may be advantageous to have a generic, low complexity multiplexing tool that allows interleaving of data with low overhead and low delay. This is particularly important for low bit rate applications. Such a multiplex tool is specified in this Subclause. Its use is optional.

7.4.2 M4Mux Tool

7.4.2.1 Overview

The M4Mux tool is a flexible multiplexer that accommodates interleaving of SL-packetized streams with varying instantaneous bit rate. The basic data entity of the M4Mux is a M4Mux packet, which has a variable length. One or more SL packets are embedded in a M4Mux packet as specified in detail in the remainder of this Subclause. The M4Mux tool provides identification of SL packets originating from different elementary streams by means of M4Mux Channel numbers. Each SL-packetized stream is mapped into one M4Mux Channel. M4Mux packets with data from different SL-packetized streams can therefore be arbitrarily interleaved. The sequence of M4Mux packets that are interleaved into one stream are called a M4Mux Stream.

A M4Mux Stream retrieved from storage or transmission may be parsed as a single data stream. However, framing of M4Mux packets by the underlying layer is required for random access or error recovery. There is no requirement to frame each individual M4Mux packet. The M4Mux also requires reliable error detection by the underlying layer. This design has been chosen acknowledging the fact that framing and error detection mechanisms are in many cases provided by the transport protocol stack below the M4Mux.

Two different modes of operation of the M4Mux providing different features and complexity are defined. They are called Simple Mode and MuxCode Mode. A M4Mux Stream may contain an arbitrary mixture of M4Mux packets using either Simple Mode or MuxCode Mode. The syntax and semantics of both modes are specified below.

The delivery timing of the M4Mux Stream can be conveyed by means of M4Mux clock reference time stamps. This functionality may be used to establish a multiplex buffer model on the delivery layer. Both the time stamps and the MuxCode Mode require out-of-band configuration prior to usage.

7.4.2.2 Simple Mode

In the simple mode one SL packet is encapsulated in one M4Mux packet and tagged by an `index` which is equal to the M4Mux Channel number as indicated in Figure 11. This mode does not require any configuration or maintenance of state by the receiving terminal.

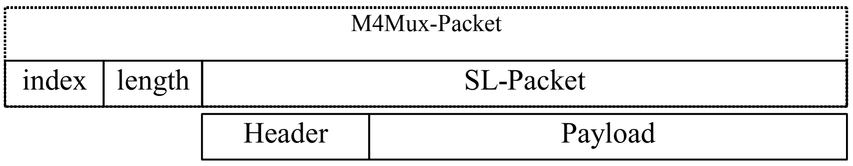


Figure 11 — Structure of M4Mux packet in simple mode

7.4.2.3 MuxCode mode

In the MuxCode mode one or more SL packets are encapsulated in one M4Mux packet as indicated in Figure 12. This mode requires configuration and maintenance of state by the receiving terminal. The configuration describes how M4Mux packets are shared between multiple SL packets. In this mode the `index` value is used to dereference configuration information that defines the allocation of the M4Mux packet payload to different M4Mux Channels.

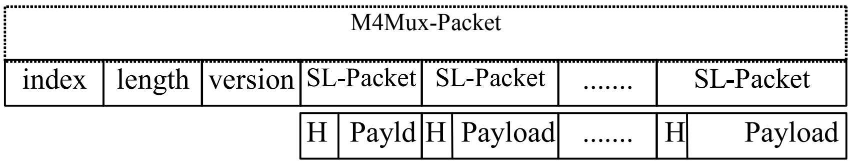


Figure 12 — Structure of M4Mux packet in MuxCode mode

7.4.2.4 M4Mux packet specification

7.4.2.4.1 Syntax

```
class M4MuxPacket (MuxCodeTableEntry mct[],
                   M4MuxTimingDescriptor FM,
                   M4MuxIDDescriptor mde) {
  unsigned int(8) index;
  if (mde == NULL | mde.Muxtype == 0) {
    bit(8) length;
  } else if (mde.Muxtype == 1) {
    length = 0;
    bit(1) nextByte;
    bit(7) length;
    while(nextByte) {
      bit(1) nextByte;
      bit(7) sizeByte;
      length = length<<7 | sizeByte;
    }
   }
    if (index<238) {
      if (length!=0) {
        SL_Packet sPayload;
        } else {
          bit(5) FMC_version_number;
          const bit(3) reserved=0b111;
        }
    } else if (index == 238) {
      bit(FM.FCR_Length) fmxClockReference;
      bit(FM.fmxRateLength) fmxRate;
        for (i=0; i<(length-FM.FCR_Length-FM.fmxRateLength); i++) {
          M4Mux_descriptor()
        }
      } else if (index == 239) {
      bit(8) stuffing[length];
    } else {
      bit(4) version;
      const bit(4) reserved=0b1111;
      multiple_SL_Packet mPayload(mct[index-240]);
    }
}
```

7.4.2.4.2 Semantics

`length` – the length of the M4Mux packet `payload` in bytes. This is equal to the length of the single encapsulated SL packet in Simple Mode and to the total length of the multiple encapsulated SL packets in MuxCode Mode. If the M4MuxIDDescriptor is not used, or if it is used and if the Muxtype is designing the first M4Mux tool, the length field is on one byte. If the M4MuxIDDescriptor is used and if the Muxtype is designing the second M4Mux tool, the length calculation relies on the combination of the `nextByte` and `sizeByte` fields that can be spread over several bytes. In Simple Mode, when this length is equal to zero, the M4Mux packet carries one byte that contains the `FMC_version_number` field. In Simple Mode, M4Mux packets with a length equal to zero (each carrying a `FMC_version_number`) can be duplicated.

`FMC_version_number` – This 5 bit field indicates the current version of the M4MuxChannelDescriptor that is applicable. `FMC_version_number` is used for error resilience purposes. If this version number does not match the version of the referenced M4MuxChannelDescriptor that has most recently been received, the following M4Mux packets belonging to the same M4Mux Channel cannot be parsed. The implementation is free to either wait until the required version of M4MuxChannelDescriptor becomes available or to discard the following M4Mux packets belonging to the same M4Mux Channel. In Simple Mode, the value given to the `FMC_version_number` field is identical in subsequent duplicated M4Mux packets with a length equal to zero.

7.4.2.5 Configuration and usage of MuxCode Mode

7.4.2.5.1 Syntax

```
aligned(8) class MuxCodeTableEntry {
   int    i, k;
   bit(8) length;
   bit(4) MuxCode;
   bit(4) version;
   bit(8) substructureCount;
   for (i=0; i<substructureCount; i++) {
      bit(5) slotCount;
      bit(3) repetitionCount;
      for (k=0; k<slotCount; k++){
         bit(8) m4MuxChannel[[i]][[k]];
         bit(8) numberOfBytes[[i]][[k]];
      }
   }
}
```

7.4.2.5.2 Semantics

The configuration for MuxCode Mode is signaled by `MuxCodeTableEntry` messages. The transport of the `MuxCodeTableEntry` shall be defined during the design of the transport protocol stack that makes use of the M4Mux tool. Part 6 of this Final Committee Draft of International Standard defines a method to convey this information using the DN_TransmuxConfig primitive.

The basic requirement for the transport of the configuration information is that data arrives reliably in a timely manner. However, no specific performance bounds are required for this control channel since version numbers allow to detect M4Mux packets that cannot currently be decoded and, hence, trigger suitable action in the receiving terminal.

`length` – the length in bytes of the remainder of the `MuxCodeTableEntry` following the `length` element.

`MuxCode` – the number through which this MuxCode table entry is referenced.

`version` – indicates the version of the `MuxCodeTableEntry`. Only the latest received version of a `MuxCodeTableEntry` is valid.

`substructureCount` – the number of substructures of this `MuxCodeTableEntry`.

`slotCount` – the number of slots with data from different M4Mux Channels that are described by this substructure.

`repetitionCount` – indicates how often this substructure is to be repeated. A `repetitionCount` zero indicates that this substructure is to be repeated infinitely. `repetitionCount` zero is only permitted in the last substructure of a MuxCodeTableEntry.

`M4MuxChannel[i][k]` – the M4Mux Channel to which the data in this slot belongs.

`numberOfBytes[i][k]` – the number of data bytes in this slot associated to `m4MuxChannel[i][k]`. This number of bytes corresponds to one SL packet.

7.4.2.5.3 Usage

The `MuxCodeTableEntry` describes how a M4Mux packet is partitioned into slots that carry data from different M4Mux Channels. This is used as a template for parsing M4Mux packets. If a M4Mux packet is longer than the template, parsing shall resume from the beginning of the template. If a M4Mux packet is shorter than the template, the remainder of the template is ignored.

Note that the usage of MuxCode mode may not be efficient if SL packets for a given elementary stream do not have a constant length. Given the overhead for an update of the associated MuxCodeTableEntry, usage of simple mode might be more efficient.

Note further that data for a single M4Mux channel may be conveyed through an arbitrary sequence of M4Mux packets with both simple mode and MuxCode mode.

EXAMPLE —

In this example we assume the presence of three substructures. Each one has a different slot count as well as repetition count. The exact parameters are as follows:

`substructureCount` = 3

`slotCount`[i] = 2, 3, 2 (for the corresponding substructure)

`repetitionCount`[i] = 3, 2, 1 (for the corresponding substructure)

We further assume that each slot configures channel number FMC*n* (`m4MuxChannel`) with a number of bytes Bytes*n* (`numberOfBytes`). This configuration would result in a splitting of the M4Mux packet payload to:

FMC1 (Bytes1), FMC2 (Bytes2)	repeated 3 times, then
FMC3 (Bytes3), FMC4 (Bytes4), FMC5 (Bytes5)	repeated 2 times, then
FMC6 (Bytes6), FMC7 (Bytes7)	repeated once

The layout of the corresponding M4Mux packet would be as shown in Figure 13.

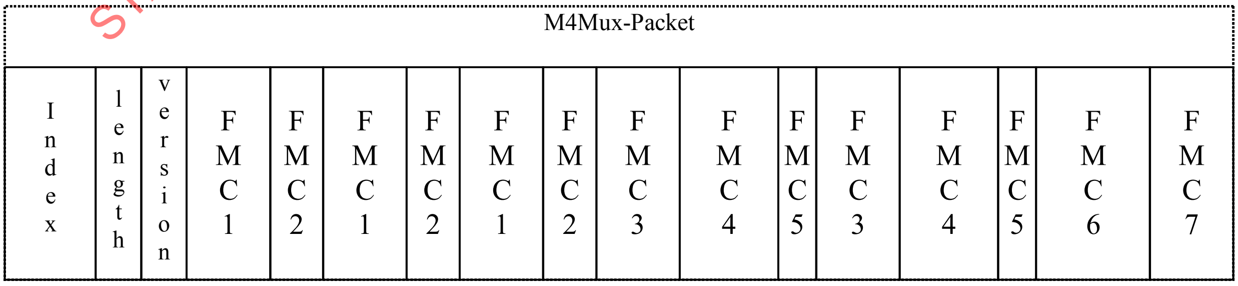


Figure 13 — Example for a M4Mux packet in MuxCode mode

7.4.2.6 Configuration and usage of M4Mux clock references

7.4.2.6.1 Syntax

```
aligned(8) class M4MuxTimingDescriptor {
   bit(16) FCR_ES_ID;
   bit(32) FCRResolution;
   bit(8) FCRLength;
   bit(8) FmxRateLength;
}
```

7.4.2.6.2 Semantics

The sequence of `fmxClockReference` time stamps in a M4Mux stream constitutes a clock reference stream, albeit with a different syntax as specified in 7.3. Elementary streams shall be associated to the time base established by this clock reference by referencing the `FCR_ES_ID` as their `OCR_ES_ID` in the `SLConfigDescriptor`. The transport of the `M4MuxTimingDescriptor` shall be defined during the design of the transport protocol stack that makes use of the M4Mux tool.

7.4.2.6.3 Usage

The M4Mux clock reference time stamps may be used to establish and verify a multiplex buffer model. The `fmxClockReference` information determines the arrival time t(i) of individual bytes i of the M4Mux stream in the following way:

$$t(i) = \frac{FCR(i'')}{FCR\,\mathrm{Re}\,solution} + \frac{i - i''}{fmxRate(i)}$$

where:

i	is the index of any byte in the M4Mux stream for i" < i < i'
i"	is the index of the byte containing the last bit of the most recent `fmxClockReference` field in the M4Mux stream

FCR(i") is the time encoded in the `fmxClockReference` in units of `FCRResolution`

fmxRate(i) indicates the rate specified by the fmxRate field for byte i

7.4.2.7 M4Mux buffer descriptor

7.4.2.7.1 Syntax

```
aligned(8) class M4MuxBufferDescriptor {
   bit(8) m4MuxChannel;
   bit(24) FB_BufferSize;
}
```

7.4.2.7.2 Semantics

The size of multiplex buffers for each M4Mux channel is signaled by `M4MuxBufferDescriptors`. One descriptor per M4Mux channel is required unless the `DefaultM4MuxBufferDescriptor` is used. The transport of the `M4MuxBufferDescriptors` shall be defined during the design of the transport protocol stack that makes use of the M4Mux tool.

`m4MuxChannel` - the number of a M4Mux channel

`FB_BufferSize` - the size of the M4Mux buffer for this M4Mux channel in bytes.

7.4.2.8 Default M4Mux buffer descriptor

7.4.2.8.1 Syntax

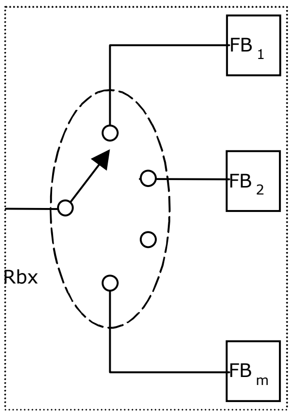
```
aligned(8) class DefaultM4MuxBufferDescriptor {
   bit(24) FB_DefaultBufferSize;
}
```

7.4.2.8.2 Semantics

The default size of multiplex buffers for each individual channel in a M4Mux stream is signaled by the `DefaultM4MuxBufferDescriptor`. M4Mux channels that use a different buffer size may signal this using the `M4MuxBufferDescriptor`. The transport of the `DefaultM4MuxBufferDescriptor` shall be defined during the design of the transport protocol stack that makes use of the M4Mux tool.

`FB_DefaultBufferSize` - the default size of M4Mux buffers for this M4Mux stream in bytes.

7.4.2.9 M4Mux buffer model



FB_n	is the M4Mux buffer for the elementary stream in M4Mux channel n
Rbx	is the rate at which data enters the M4Mux buffers.

The M4Mux buffer model applies to M4Mux streams that utilize M4Mux Clock reference channel packets to define the delivery timing of the M4Mux stream. The M4Mux stream enters the M4Mux buffer model at the rate and timing as defined by the fmxClockReference and fmxRate fields. There may be some periods of time during which there are no bytes at the input of the M4Mux buffer model, but the bytes of all M4Mux packets that preceed the next M4Mux Clock reference channel packet shall be delivered to the M4Mux buffer model prior to the delivery of any byte of the next M4Mux Clock reference channel packet.

For each M4Mux channel i the M4Mux packet is stored in M4Mux Buffer FB_i. The bytes in buffer FBi are removed at a rate specified by the InstantRate field in the SL header of the contained SL-packetized stream. Upon removal each byte enters the elementary stream buffer DB_i. The M4Mux stream shall be constructed so that the following condition is met :

- Buffer FBi shall not overflow.

7.4.2.10 M4MuxID Descriptor

7.4.2.10.1 Syntax

```
aligned(8) class M4MuxIDDescriptor {
   bit(8) MuxID;
   bit(4) Muxtype;
    bit(4) Muxmanagement;
}
```

7.4.2.10.2 Semantics

`MuxID` – the ID of the M4Mux stream.

`Muxtype` – the type of the Multiplexing tool used to generate the M4Mux stream. Indicated type value shall comply with the following Table 16 — Multiplexing type table.

`Muxmanagement` – the mode of management used by the Multiplexing tool, to generate the M4Mux stream. Indicated mode value shall comply with Table 17 — Multiplexing management mode table.

Table 16 — Multiplexing type table

Type	Multiplexing tool
0	M4Mux tool
1	M4Mux_2 tool
2-7	ISO/IEC 14496-1 Reserved
8-15	User Private

Table 17 — Multiplexing management mode table

Type	management mode
0	Static
1	Dynamic
2-7	ISO/IEC 14496-1 Reserved
8-15	User Private

7.4.3 M4Mux Descriptors

Directly derived from the M4Mux descriptor classes, hereafter are defined the M4Mux descriptors pointed to by the “List of Class Tags for Descriptors” table.

7.4.3.1 M4MuxChannelDescriptor

7.4.3.1.1 Syntax

```
class M4MuxChannelDescriptor extends BaseDescriptor
            : bit(8) tag= M4MuxChannelDescrTag {
     bit(5) version_number;
     bit(1) current_next_indicator;
     const bit(2) reserved=0b11;
     for (i=0; i<( sizeOfInstance-2); i += 3) {
     bit(16) ES_ID;
     bit(8) M4MuxChannel;
     }
}
```

7.4.3.1.2 Semantics

version_number -- This 5 bit field is the version number of the complete M4MuxChannelDescriptor. The version number shall be incremented by 1 whenever the definition of the M4MuxChannelDescriptor changes. Upon reaching the value 31, it wraps around to 0. When the current_next_indicator is set to '1', then the version_number shall be that of the currently applicable M4MuxChannelDescriptor. When the current_next_indicator is set to '0', then the version_number shall be that of the next applicable M4MuxChannelDescriptor.

current_next_indicator -- A 1 bit indicator, which when set to '1' indicates that the received M4MuxChannelDescriptor is currently applicable. When the bit is set to '0', it indicates that the received M4MuxChannelDescriptor is not yet applicable and shall be the next M4MuxChannelDescriptor to become valid.

A validity period of time is associated with each version_number of a M4MuxChannelDescriptor. It is only within that validity period of time, that M4Mux packets refer to the version identified by that version_number. The validity period of time of one version starts as soon as the first M4MuxChannelDescriptor is sent with the current_next_indicator == 1.

The validity period of time of one version ends as soon as an empty M4MuxChannelDescriptor is sent with the current_next_indicator == 1, meaning that the assignements of that version of the M4MuxChannelDescriptor are not any more relevant.

An empty M4MuxChannelDescriptor is a M4MuxChannelDescriptor shall be sent with sizeOfInstance == 1, such that there are no elementary streams described.

ES_ID – this 16-bit field specifies the identifier of an ISO/IEC 14496-1 SL-packetized stream.

M4MuxChannel - This 8-bit field specifies the number of the M4Mux channel used for this SL-packetized stream.

7.4.3.2 M4MuxBufferSize Descriptor

7.4.3.2.1 Syntax

```
class M4MuxBufferSizeDescriptor extends BaseDescriptor
            : bit(8) tag= M4MuxBufferSizeDescrTag {
            DefaultM4MuxBufferDescriptor()
            for (i=0; i<( sizeOfInstance-3); i += 4) {
                   M4MuxBufferDescriptor()
            }
}
```

7.4.3.2.2 Semantics

DefaultM4MuxBufferDescriptor - the default size of multiplex buffers for each individual channel in a M4Mux stream is signalled by the DefaultM4MuxBufferDescriptor class.

M4MuxBufferDescriptor - the exact size of multiplex buffers for each channel in a M4Mux stream can be signalled by the M4MuxBufferDescriptor class.

7.4.3.3 M4MuxTiming Descriptor

7.4.3.3.1 Syntax

```
class M4MuxTimingDescriptor extends BaseDescriptor
            : bit(8) tag= M4MuxTimingDescrTag {
            M4MuxTimingDescriptor()
}
```

7.4.3.3.2 Semantics

M4MuxTimingDescriptor – This descriptor class defines FCR_ES_ID, FCRResolution, FCRLength, FmxRateLength.

7.4.3.4 M4MuxCodeTable Descriptor

7.4.3.4.1 Syntax

```
class M4MuxCodeTableDescriptor extends BaseDescriptor
       : bit(8) tag= M4MuxCodeTableDescrTag {
            for(i =0; i < sizeOfInstance; i += sizeof ( MuxCodeTableEntry () ) )
            {
       MuxCodeTableEntry ()
       }
}
```

7.4.3.4.2 Semantics

MuxCodeTableEntry () – This class defines the M4Mux configuration of one M4Mux channel.

Several M4MuxCodeTableDescriptor may be used with different instances of the MuxCodeTableEntry class.

7.4.3.5 M4MuxIdent Descriptor

7.4.3.5.1 Syntax

```
class M4MuxIdentDescriptor extends BaseDescriptor
            : bit(8) tag= M4MuxIdentDescrTag {
            M4MuxIDDescriptor ()
}
```

7.4.3.5.2 Semantics

M4MuxIDDescriptor – This class defines MuxID, Muxtype, Muxmanagement.

8 Syntactic Description Language

8.1 Introduction

This Subclause describes the mechanism with which bitstream syntax is documented in ISO/IEC 14496. This mechanism is based on a Syntactic Description Language (SDL), documented here in the form of syntactic description rules. It directly extends the C-like syntax used in ISO/IEC 11172-1:1993 and ISO/IEC 13818-1:2007 into a well-defined framework that lends itself to object-oriented data representations. In particular, SDL assumes an object-oriented underlying framework in which bitstream units consist of “classes.” This framework is based on the typing system of the C++ and Java programming languages. SDL extends the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed.

The elementary constructs are described first, followed by the composite syntactic constructs, and arithmetic and logical expressions. Finally, syntactic control flow and built-in functions are addressed. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure.

8.2 Elementary Data Types

8.2.1 Introduction

The SDL uses the following elementary data types:

1. Constant-length direct representation bit fields or Fixed Length Codes — FLCs. These describe the encoded value exactly as it is to be used by the appropriate decoding process.
2. Variable length direct representation bit fields, or parametric FLCs. These are FLCs for which the actual length is determined by the context of the bitstream (e.g., the value of another parameter).
3. Constant-length indirect representation bit fields. These require an extra lookup into an appropriate table or variable to obtain the desired value or set of values.
4. Variable-length indirect representation bit fields (e.g., Huffman codes).

These elementary data types are described in more detail in the Clauses to follow immediately.

All quantities shall be represented in the bitstream with the most significant byte first, and also with the most significant bit first.

8.2.2 Constant-Length Direct Representation Bit Fields

Constant-length direct representation bit fields shall be represented as:

Rule E.1: Elementary Data Types

[**aligned**] ***type***[**(***length***)**] *element_name* [= *value*]; // C++-style comments allowed

The ***type*** may be any of the following: **int** for signed integer, **unsigned int** for unsigned integer, **double** for floating point, and **bit** for raw binary data. The *length* attribute indicates the length of the element in bits, as it is actually stored in the bitstream. Note that a data ***type*** equal to **double** shall only use 32 or 64 bit lengths. The *value* attribute shall be present only when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., ‘0x01..0xAF’). The ***type*** and the optional *length* attributes are always present, except if the data is non-parsable, i.e., it is not included in the bitstream. The keyword **aligned** indicates that the data is aligned on a byte boundary. As an example, a start code would be represented as:

```
aligned bit(32) picture_start_code=0x00000100;
```

An optional numeric modifier, as in **aligned**(32), may be used to signify alignment on other than byte boundary. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value ‘0’. An entity such as temporal reference would be represented as:

```
unsigned int(5) temporal_reference;
```

where **unsigned int**(*5*) indicates that the element shall be interpreted as a 5-bit unsigned integer. By default, data shall be represented with the most significant bit first, and the most significant byte first.

The value of parsable variables with declarations that fall outside the flow of declarations shall be set to 0.

Constants shall be defined using the keyword **const**.

EXAMPLE —

```
const int SOME_VALUE=255; // non-parsable constant
const bit(3) BIT_PATTERN=1;  // this is equivalent to the bit string "001"
```

To designate binary values, the **0b** prefix shall be used, similar to the **0x** prefix for hexadecimal numbers. A period ('.') may be optionally placed after every four digits for readability. Hence 0x0F is equivalent to 0b0000.1111.

In several instances, it may be desirable to examine the immediately following bits in the bitstream, without actually consuming these bits. To support this behavior, a '*****' character shall be placed after the parse size parentheses to modify the parse size semantics.

Rule E.2: Look-ahead parsing

[**aligned**] ***type*** **(***length***)*** *element_name*;

For example, the value of next 32 bits in the bitstream can be checked to be an unsigned integer without advancing the current position in the bitstream using the following representation:

```
aligned unsigned int (32)* next_code;
```

8.2.3 Variable Length Direct Representation Bit Fields

This case is covered by Rule E.1, by allowing the *length* attribute to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables.

EXAMPLE —

```
unsigned int(3) precision;
int(precision) DC;
```

8.2.4 Constant-Length Indirect Representation Bit Fields

Indirect representation indicates that the actual value of the element at hand is indirectly specified by the bitstream through the use of a table or map. In other words, the value extracted from the bitstream is an index to a table from which the final desired value is extracted. This indirection may be expressed by defining the map itself:

Rule E.3: Maps

```
map MapName (output_type) {
   index, {value_1, … value_M},
   …
}
```

These tables are used to translate or map bits from the bitstream into a set of one or more values. The input type of a **map** (the *index* specified in the first column) shall always be **bit**. The *output_type* entry shall be either a predefined type or a defined class (classes are defined in 8.3.1). The **map** is defined as a set of pairs of such indices and values. Keys are binary string constants while values are *output_type* constants. Values shall be specified as aggregates surrounded by curly braces, similar to C or C++ structures.

EXAMPLE —

```
class YUVblocks {// classes are fully defined later on
   int Yblocks;
   int Ublocks;
   int Vblocks;
}

// a table that relates the chroma format with the number of blocks
// per signal component
map blocks_per_component (YUVblocks) {
   0b00,{4, 1, 1},// 4:2:0
   0b01,{4, 2, 2},// 4:2:2
   0b10,{4, 4, 4} // 4:4:4
}
```

The next rule describes the use of such a **map**.

Rule E.4: Mapped Data Types

type **(***MapName***)** *name*;

The ***type*** of the variable shall be identical to the ***type*** returned from the **map**.

EXAMPLE —

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, a particular value of the **map** may be accessed using the construct: `chroma_format.Ublocks`.

8.2.5 Variable Length Indirect Representation Bit Fields

For a variable length element utilizing a Huffman or variable length code table, an identical specification to the fixed length case shall be used:

```
class val {
   unsigned int foo;
   int bar;
}

map sample_vlc_map (val)  {
   0b0000.001,   {0, 5},
   0b0000.0001,  {1, -14}
}
```

The only difference is that the indices of the **map** are now of variable length. The variable-length codewords are (as before) binary strings, expressed by default in ‘0b’ or ‘0x’ format, optionally using the period (‘.’) every four digits for readability.

Very often, variable length code tables are partially defined. Due to the large number of possible entries, it may be inefficient to keep using variable length codewords for all possible values. This necessitates the use of escape codes, that signal the subsequent use of a fixed-length (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for **map** values.

EXAMPLE — This example uses the class type 'val' as defined above.

```
map sample_map_with_esc (val)  {
   0b0000.001,     {0, 5},
   0b0000.0001, {1, -14},
   0b0000.0000.1,  {5, int(32)},
   0b0000.0000.0,  {0, -20}
}
```

When the codeword 0b0000.0000.1 is encountered in the bitstream, then the value '5' is assigned to the first element (`val.foo`). The following 32 bits are parsed and assigned as the value of the second element (`val.bar`). Note that, in case more than one element utilizes a parsable type declaration, the order is significant and is the order in which elements are parsed. In addition, the type within the **map** declaration shall match the type used in the class declaration associated with the **map**'s return type.

8.3 Composite Data Types

8.3.1 Classes

Classes are the mechanism with which definitions of composite types or objects is performed. Their definition is as follows.

Rule C.1: Classes

[**aligned**] [**abstract**] [**expandable**[**(***maxClassSize***)**]] **class** *object_name* [**extends** *parent_class*]
[: **bit(***length***)** [*id_name*=] *object_id* | *id_range* | *extended_id_range*] **{**
[*element*; …] // zero or more elements
}

The different elements within the curly braces are the definitions of the elementary bitstream components discussed in 12.2 or control flow elements that will be discussed in a subsequent Subclause.

The optional keyword **extends** specifies that the **class** is "derived" from another **class**. Derivation implies that all information present in the base **class** is also present in the derived **class**, and that, in the bitstream, all such information *precedes* any additional bitstream syntax declarations specified in the new **class**.

The optional attribute *id_name* allows to assign an *object_id*, and, if present, is the key demultiplexing entity which allows differentiation between base and derived objects. It is also possible to have a range of possible values: the *id_range* is specified as *start_id* .. *end_id*, inclusive of both bounds. It is also possible to have a combination of *id_range* and *object_id:* the *extended_id_range* is specified as a comma-separated list of *object_id* and *range_id*; for example, *id_name=object_id1*, *object_id2*, *start_id* .. *end_id.*

If the attribute id_name is used, a derived **class** may appear at any point in the bitstream where its base **class** is specified in the syntax. This allows to express polymorphism in the SDL syntax description. The actual **class** to be parsed is determined as follows:

- The base **class** declaration shall assign a constant value or range of values to *object_id*.
- Each derived **class** declaration shall assign a constant value or ranges of values to *object_id*. This value or set of values shall correspond to legal *object_id* value(s) for the base **class**.

NOTE 1 — Derivation of classes is possible even when object_ids are not used. However, in that case derived classes may not replace their base **class** in the bitstream.

NOTE 2 — Derived classes may use the same *object_id* value as the base **class**. In that case classes can only be discriminated through context information.

EXAMPLE —

```
class slice: aligned bit(32) slice_start_code=0x00000101 .. 0x000001AF {
   // here we get vertical_size_extension, if present
   if (scalable_mode==DATA_PARTITIONING) {
      unsigned int(7) priority_breakpoint;
   }
   …
}

class foo {
   int(3) a;
   ...
}

class bar extends foo {
   int(5) b; // this b is preceded by the 3 bits of a
   int(10) c;
   ...
}
```

The order of declaration of the bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, `bar.b` immediately precedes `bar.c` in the bitstream.

Objects may also be encapsulated within other objects. In this case, the *element* in Rule C.1 is an object itself.

8.3.2 Abstract Classes

When the **abstract** keyword is used in the **class** declaration, it indicates that only derived classes of this **class** shall be present in the bitstream. This implies that the derived classes may use the entire range of IDs available. The declaration of the abstract **class** requires a declaration of an ID, with the value 0.

EXAMPLE —

```
abstract class Foo : bit(1) id=0 { // the value 0 is not really used
   ...
}

// derived classes are free to use the entire range of IDs
class Foo0 extends Foo : bit(1) id=0 {
   ...
}

class Foo1 extends Foo : bit(1) id=1 {
   ...
}

class Example {
   Foo f;  // can only be Foo0 or Foo1, not Foo
}
```

8.3.3 Expandable classes

When the **expandable** keyword is used in the **class** declaration, it indicates that the **class** may contain implicit arrays or undefined trailing data, called the "expansion". In this case the **class** encodes its own size in bytes explicitly. This may be used for classes that require future compatible extension or that may include private data. A legacy device is able to decode an expandable **class** up to the last parsable variable that has been defined for a given revision of this **class**. Using the size information, the parser shall skip the **class** data following the last known syntax element. Anywhere in the syntax where a set of expandable classes with *object_id* is expected it is permissible to intersperse expandable classes with unknown *object_id* values. These classes shall be skipped, using the size information.

The size encoding precedes any parsable variables of the **class**. If the **class** has an *object_id*, the encoding of the *object_id* precedes the size encoding. The size information shall not include the number of bytes needed for the size and the *object_id* encoding. Instances of expandable classes shall always have a size corresponding to an integer number of bytes. The size information is accessible within the class as class instance variable `sizeOfInstance`.

If the **expandable** keyword has a *maxClassSize* attribute, then this indicates the maximum permissible size of this **class** in bytes, including any expansion.

The length encoding is itself defined in SDL as follows:

```
int sizeOfInstance = 0;
bit(1) nextByte;
bit(7) sizeOfInstance;
while(nextByte) {
   bit(1) nextByte;
   bit(7) sizeByte;
   sizeOfInstance = sizeOfInstance<<7 | sizeByte;
}
```

8.3.4 Parameter types

A parameter type defines a **class** with parameters. This is to address cases where the data structure of the **class** depends on variables of one or more other objects. Since SDL follows a declarative approach, references to other objects, in such cases, cannot be performed directly (none is instantiated). Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a **class** definition with parameters is as follows.

Rule C.2: Class Parameter Types

[**aligned**] [**abstract**] **class** *object_name* [**(***parameter list***)**] [**extends** *parent_class*]
[: **bit(***length***)** [*id_name*=] *object_id* | *id_range*] **{**
[*element*; …] // zero or more elements
}

The parameter list is a list of ***type*** names and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a variable-length codeword, or a constant, can be passed as a parameter.

A **class** that uses parameter types is dependent on the objects in its parameter list, whether **class** objects or simple variables. When instantiating such a **class** into an object, the parameters have to be instantiated objects of their corresponding classes or types.

EXAMPLE —

```
class A {
   // class body
   ...
   unsigned int(4) format;
}

class B (A a, int i) {     // B uses parameter types
   unsigned int(i) bar;
   ...
   if( a.format == SOME_FORMAT ) {
      ...
   }
   ...
}
```

```
class C {
   int(2) i;
   A a;
   B foo( a, I); // instantiated parameters are required
}
```

8.3.5 Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. The array declaration is applicable to both elementary as well as composite objects.

Rule A.1: Arrays

typespec *name* **[***length***]**;

typespec is a ***type*** specification (including bitstream representation information, e.g. '**int(2)**'). The attribute *name* is the name of the array, and *length* is its length.

EXAMPLE —

```
unsigned int(4) a[5];
int(10) b;
int(2) c[b];
```

Here 'a' is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer. In the case of 'c', its length depends on the actual value of 'b'. Multi-dimensional arrays are allowed as well. The parsing order from the bitstream corresponds to scanning the array by incrementing first the right-most index of the array, then the second, and so on .

8.3.6 Partial Arrays

In several situations, it is desirable to load the values of an array one by one, in order to check, for example, a terminating or other condition. For this purpose, an extended array declaration is allowed in which individual elements of the array may be accessed.

Rule A.2: Partial Arrays

typespec *name***[[***index***]]**;

Here *index* is the element of the array that is defined. Several such partial definitions may be given, but they shall all agree on the ***type*** specification. This notation is also valid for multidimensional arrays.

EXAMPLE —

```
int(4) a[[3]][[5]];
```

indicates the element a(5, 3) of the array (the element in the 6th row and the 4th column), while

```
int(4) a[3][[5]];
```

indicates the entire sixth column of the array, and

```
int(4) a[[3]][5];
```

indicates the entire fourth row of the array, with a length of 5 elements.

NOTE — **a[*5*]** means that the array has five elements, whereas **a[[*5*]]** implies that there are at least six.

8.3.7 Implicit Arrays

When a series of polymorphic classes is present in the bitstream, it may be represented as an array of the same type as that of the base **class**. Let us assume that a set of polymorphic classes is defined, derived from the base **class** Foo (may or may not be abstract):

```
class Foo : int(16) id = 0 {
   ...
}
```

For an array of such objects, it is possible to implicitly determine the length by examining the validity of the **class** ID. Objects are inserted in the array as long as the ID can be properly resolved to one of the IDs defined in the base (if not abstract) or its derived classes. This behavior is indicated by an array declaration without a length specification.

EXAMPLE 1 —

```
class Example {
   Foo f[];   // length implicitly obtained via ID resolution
}
```

To limit the minimum and maximum length of the array, a range specification may be inserted in the specification of the length.

EXAMPLE 2 —

```
class Example {
   Foo f[1 .. 255];   // at least 1, at most 255 elements
}
```

In this example, ‘f’ may have at least 1 and at most 255 elements.

8.4 Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C++ are allowed, including their precedence rules.

8.5 Non-Parsable Variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but only as a result of a non-trivial computation, non-parsable variables are allowed. These are strictly of local scope to the **class** they are defined in. They may be used in expressions and conditions in the same way as bitstream-level variables. In the following example, the number of non-zero elements of an array is computed.

```
unsigned int(6) size;
int(4) array[size];
…
int i; // this is a temporary, non-parsable variable
for (i=0, n=0; i<size; i++) {
   if (array[[i]]!=0)
      n++;
}

int(3) coefficients[n];
// read as many coefficients as there are non-zero elements in array
```

8.6 Syntactic Flow Control

The syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C/C++ if-then-else construct is used for testing conditions. Similarly to C/C++, zero corresponds to false, and non-zero corresponds to true.

Rule FC.1: Flow Control Using If-Then-Else

```
if (condition) {
   …
} [else if (condition) {
   …
}][else {
   …
}]
```

EXAMPLE 1 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   }
   unsigned int(32) more_foo;
}
```

Here the presence of the entity 'bar' is determined by the 'bar_flag'.

EXAMPLE 2 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   } else {
      unsigned int(some_vlc_table) bar;
   }
   unsigned int(32) more_foo;
}
```

Here we allow two different representations for 'bar', depending on the value of 'bar_flag'. We could equally well have another entity instead of the second version (the variable length one) of 'bar' (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered. Also, if a variable appears twice (as in the example above), the types shall be identical.

In order to facilitate cascades of if-then-else constructs, the 'switch' statement is also allowed.

Rule FC.2: Flow Control Using Switch

```
switch (condition) {
      [case label1: …]
      [default:]
}
```

The same category of context-sensitive objects also includes iterative definitions of objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of 'for', 'while', and 'do' loops can be used for this purpose.

Rule FC.3: Flow Control Using For

```
for   (expression1; expression2; expression3) {
      …
}
```

expression1 is executed prior to starting the repetitions. Then *expression2* is evaluated, and if it is non-zero (true) the declarations within the braces are executed, followed by the execution of *expression3*. The process repeats until *expression2* evaluates to zero (false).

Note that it is not allowed to include a variable declaration in *expression1* (in contrast to C++).

Rule FC.4: Flow Control Using Do

```
do {
      …
} while (condition);
```

Here the block of statements is executed until *condition* evaluates to false. Note that the block will be executed at least once.

Rule FC.5: Flow Control Using While

```
while (condition) {
      …
}
```

The block is executed zero or more times, as long as *condition* evalutes to non-zero (true).

8.7 Built-In Operators

The following built-in operators are defined.

Rule O.1: lengthof() Operator

```
lengthof(variable)
```

This operator returns the length, in bits, of the quantity contained in parentheses. The length is the number of bits that was most recently used to parse the quantity at hand. A return value of 0 means that no bits were parsed for this variable.

8.8 Scoping Rules

All parsable variables have class scope, i.e., they are available as class member variables.

For non-parsable variables, the usual C++/Java scoping rules are followed (a new scope is introduced by curly braces: '{' and '}'). In particular, only variables declared in class scope are considered class member variables, and are thus available in objects of that particular type.

9 Profiles

9.1.1 Introduction

This Subclause defines profiles and levels for the usage of the tools defined in this part of ISO/IEC 14496. Each profile at a given level constitutes a subset of this part of ISO/IEC 14496 to which system manufacturers and content creators can claim conformance in order to ensure interoperability.

The object descriptor profiles (OD profiles) specify the allowed configurations of the object descriptor tool and the sync layer tool.

Profile definitions, by themselves, are not sufficient to provide a full characterization of a receiving terminal's capabilities and the resources needed for a presentation. For this reason, levels are defined within each profile. Levels constrain the values of parameters in a given profile in order to specify an upper complexity bound.

9.1.2 OD Profile Definitions

9.1.2.1 Overview

The object descriptor profiles (OD profiles) specify the configurations of the object descriptor tool and the sync layer tool that are allowed. The object descriptor tool provides a structure for all descriptive information. The sync layer tool provides the syntax to convey, among others, timing information for elementary streams. object descriptor profiles are used, in particular, to reduce the amount of asynchronous operations as well as the amount of permanent storage.

9.1.2.2 OD Profiles Tools

The following tools are available to construct OD profiles:

- Object descriptor (OD) tool as defined in 7.2.5.
- Sync layer (SL) tool as defined in 7.3.2
- Object content information (OCI) tool as defined in 7.2.4.
- Intellectual property management and protection (IPMP) tool as defined in 7.2.3.

9.1.2.3 OD Profiles

The OD profiles are defined in the following table. Currently, only one profile is defined, comprising all the tools. No additional profiles are foreseen at the moment, but the possibility of adding Profiles through amendments is left open.

Table 18 — OD Profiles

	OD Profiles
OD Tools	**Core**
SL	X
OD	X
OCI	X
IPMP	X

Decoders that claim compliance to a given profile shall implement all the tools with an 'X' entry for that profile.

9.1.2.4 OD Profiles@Levels

9.1.2.4.1 Levels for the Core Profile

No levels are defined yet for the OD Core profile. Future definition of Levels is anticipated; this will happen by means of an amendment to this part of the standard.

Annex A
(informative)

Time Base Reconstruction

A.1 Time Base Reconstruction

The time stamps present in the sync layer are the means to synchronize events related to decoding, composition and overall buffer management. In particular, the clock references are the sole means of reconstructing the sending terminal's clock at the receiving terminal, when required (e.g., for broadcast applications). A normative method for this reconstruction is not specified. The following describes the process at a conceptual level.

A.1.1 Adjusting the Receiving Terminal's OTB

Each elementary stream may be generated by an encoder at the sending terminal with a different object time base (OTB). For each stream that conveys OCR information, it is possible for the receiving terminal to adjust a local OTB to the sending terminals' OTB. This is done by using well-known PLL techniques. The notion of time for each data stream can therefore be recovered at the receiving end.

A.1.2 Mapping Time Stamps to the STB

The OTBs of all data streams may run at a different speed than the STB of the receiving terminal. Therefore, a method is needed to map the value of time stamps expressed in any OTB to the STB of the receiving terminal. This step may be done jointly with the recovery of individual OTB's as described in the previous Subclause.

Note that the receiving terminals' system time base need not be locked to any of the available object time bases.

The composition time t_{SCT} of a composition unit, expressed in terms of STB of the receiving terminal, can be calculated from the composition time stamp value t_{OCT}, expressed in terms of the OTB of the relevant sending terminal, by a linear transformation:

$$t_{SCT} = \frac{\Delta t_{STB}}{\Delta t_{OTB}} \cdot t_{OCT} - \frac{\Delta t_{STB}}{\Delta t_{OTB}} \cdot t_{OTB-START} + t_{STB-START}$$

with:

t_{SCT}	composition time of a composition unit measured in units of t_{STB}
t_{STB}	current time in the receiving terminal's STB
t_{OCT}	composition time of a composition unit measured in units of t_{OTB}
t_{OTB}	current time in the data stream's OTB, conveyed by an OCR
$t_{STB-START}$	value of receiving terminal's STB when the first byte of the OCR time stamp of the data stream is encountered

$t_{OTB-START}$ value of the first OCR time stamp of the data stream

$$\Delta t_{OTB} = t_{OTB} - t_{OTB-START}$$

$$\Delta t_{STB} = t_{STB} - t_{STB-START}$$

The quotient $\Delta t_{STB} / \Delta t_{OTB}$ is the instantaneous scaling factor between the two time bases. In cases where the clock speed and resolution of the sending terminal and of the receiving terminal are nominally identical, this quotient is very near 1. To avoid long term rounding errors, the quotient $\Delta t_{STB} / \Delta t_{OTB}$ should always be recalculated whenever the formula is applied to a newly received composition time stamp. The quotient can be updated each time an OCR time stamp is encountered.

A similar formula can be derived for decoding times by replacing composition time stamps with decoding time stamps. If time stamps for some access units or composition units are only known implicitly, e.g., given by known update rates, these have to be mapped with the same mechanism.

With this procedure it is possible to synchronize the STB at a receiving terminal to several OTBs so that correct decoding and composition from several data streams is possible.

A.1.3 Adjusting the STB to an OTB

When all data streams in a presentation use the same OTB, it is possible to lock the STB at the receiving terminal to this OTB using well-known PLL techniques. In this case the mapping described in the previous Subclause is not necessary and the following mapping may be used.

$$t_{STB-START} = t_{OTB-START}$$
$$\Delta t_{STB} = \Delta t_{OTB}$$
$$t_{SCT} = t_{OCT}$$

A.1.4 System Operation without Object Time Base

If a time base for an elementary stream is neither conveyed by OCR information nor derived from another elementary stream, time stamps can still be used by a receiving terminal but not in applications that require flow-control. For example, file-based playback may not require time base reconstruction: time stamps alone are sufficient for synchronization if a single time base is assumed for all the data streams.

In the absence of time stamps, the receiving terminal may only operate under the assumption that each access unit is to be decoded and presented as soon as it is received. In this case the systems decoder model does not apply and cannot be used as a model for the terminal's behavior.

In the case that a universal clock is available which can be shared between peer terminals, it may be used as a common time base. It is then possible to use the systems decoder model without explicit OCR transmission. The procedures for doing so are application-dependent and are not defined in ISO/IEC 14496-1.

A.2 Temporal aliasing and audio resampling

A receiving terminal compliant with ISO/IEC 14496 is not required to synchronize decoding of AUs and composition of CUs. In other words, its STB does not have to be identical to any of the OTBs of received data streams. The number of decoded and actually presented (displayed/played back) units per second may therefore differ. Temporal aliasing may then manifest itself as composition units being either presented multiple times or skipped.

If audio signals are encoded on a system with an OTB different from the STB of the receiving terminal, even nominally identical sampling rates of the audio samples may not match exactly, so that audio samples may be dropped or repeated.

Proper re-sampling techniques may of course in both cases be applied at the receiving terminal.

A.3 Reconstruction of a Synchronised Audio-visual Scene: A Walkthrough

The different steps to reconstruct a synchronized scene are as follows:

1. The time base for each data stream is recovered either from the OCR conveyed with the SL-packetized elementary stream of this data stream or from another data stream present in the presentation.

2. Object time stamps are mapped to the STB of the receiving terminal according to a suitable algorithm (e.g., the one detailed above).

3. Received access units are placed in the decoding buffer.

4. Each access unit is instantaneously decoded by the decoder at instants of time (in terms of the receiver terminal's STB) corresponding to its implicit or explicit DTS and the resulting one or more composition units are placed in the composition memory.

The compositor may access each CU at time instants between the one corresponding its CTS and the one corresponding to the CTS of the subsequent CU.