INTERNATIONAL ORGANISATION FOR STANDARDISATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND AUDIO

ISO/IEC JTC1/SC29/WG11 N2201
15 May 1998

Source:	**MPEG-4 Systems**
Status:	**Approved at the 43rd Meeting**
Title:	**Text for ISO/IEC FCD 14496-1 Systems**
Authors:	**Alexandros Eleftheriadis, Carsten Herpel, Ganesh Rajan, and Liam Ward (Editors)**

INFORMATION TECHNOLOGY –
GENERIC CODING OF AUDIO-VISUAL OBJECTS

Part 1: Systems

ISO/IEC 14496-1

Final Committee Draft of International Standard

Version of: 2 October, 2001, 16:08

Please address any comments or suggestions to: `spec-sys@fzi.de`

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Overview

The Systems part of this Final Committee Draft of International Standard describes a system for communicating interactive audiovisual scenes. Such scenes consist of:

1. the coded representation of natural or synthetic, 2D or 3D objects that can be manifested audibly and/or visually (media objects);
2. the coded representation of the spatio-temporal positioning of media objects as well as their behavior in response to interaction (scene description); and
3. the coded representation of information related to the management of information streams (synchronization, identification, description and association of stream content).

The overall operation of a system communicating such audiovisual scenes is as follows. At the sending side, audiovisual scene information is compressed, supplemented with synchronization information and passed to a delivery layer that multiplexes it in one or more coded binary streams that are transmitted or stored. At the receiver these streams are demultiplexed and decompressed. The media 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 the presentation. Interaction information can be processed locally or transmitted to the sender. This specification defines the semantic and syntactic rules of bitstreams that convey such scene information, as well as the details of their decoding processes.

In particular, the Systems part of this Final Committee Draft of International Standard specifies the following tools:

- a terminal model for time and buffer management;
- a coded representation of interactive audiovisual scene description information (Binary Format for Scenes – BIFS);
- a coded representation of identification and description of audiovisual streams as well as the logical dependencies between stream information (Object and other Descriptors);
- a coded representation of synchronization information (Sync Layer – SL);
- a multiplexed representation of individual streams in a single stream (FlexMux); and
- a coded representation of descriptive audiovisual content information (Object Content Information – OCI).

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

1.2 Architecture

The information representation specified in this Final Committee Draft of International Standard describes an interactive audiovisual scene in terms of coded audiovisual information and associated scene description information. The entity that receives and presents such a coded representation of an interactive audiovisual scene is generically referred to as an "audiovisual terminal" or just "terminal." This terminal may correspond to a standalone application or be part of an application system.

The basic operations performed by such a system are as follows. Information that allows access to content complying with this Final Committee Draft of International Standard is provided as initial session set up information to the terminal. Part 6 of this specification defines the procedures for establishing such session context as well as the interface (DAI – DMIF Application Interface) to the delivery layer that generically abstracts the storage or transport medium. The initial set up information allows in a recursive process 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 (FlexMux) described in this Final Committee Draft of International Standard.

Elementary streams contain the coded representation of the content data: audio or visual (AV) objects, scene description information (BIFS), information sent to identify streams or to describe the logical dependencies between streams (descriptors), or content related information (OCI streams). Each elementary stream contains only one type of information and may be a downchannel stream (sender to receiver) or an upchannel stream (receiver to sender).

Elementary streams are decoded using their respective stream-specific decoders. The media objects are composed according to the scene description information and presented to 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 1-1, and are described in more detail below.

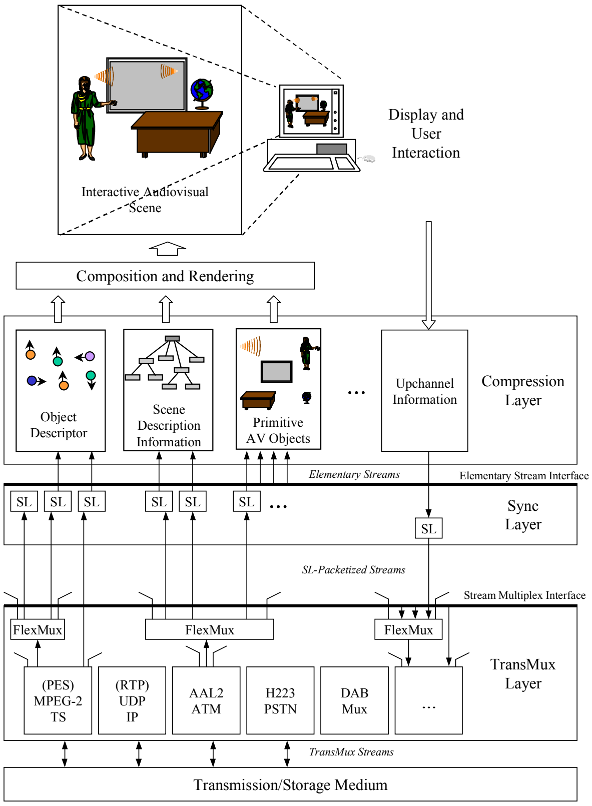


Figure 1-1: Processing stages in an audiovisual terminal

1.3 Terminal Model: Systems Decoder Model

The Systems Decoder Model provides an abstract view of the behavior of a terminal complying with this Final Committee Draft of International Standard. Its purpose is to allow a sender to predict how the receiver will behave in terms of buffer management and synchronization when reconstructing the audiovisual information that comprises the session. The Systems Decoder Model includes a timing model and a buffer model.

1.3.1 Timing Model

The Timing Model defines the mechanisms through which a receiver establishes a notion of time and performs time-dependent events. This allows the receiver to maintain synchronization both across and within particular media types as well as with user interaction events. The Timing Model requires that the transmitted data streams contain implicit or explicit timing information. Two sets of timing information are defined: clock references and time stamps. The former are used to convey the sender's time base to the receiver, while the latter convey the time (in units of a sender's time base) for specific events such as the desired decoding or composition time for portions of the encoded audiovisual information.

1.3.2 Buffer Model

The Buffer Model enables the sender to monitor and control the buffer resources that are needed to decode each individual Elementary Stream in the session. The required buffer resources are conveyed to the receiver by means of Elementary Streams Descriptors at the beginning of the session, so that it can decide whether or not it is capable of handling this particular session. The model allows the sender to specify when information is removed from these buffers and schedule data transmission so that overflow does not occur.

1.4 Multiplexing of Streams: TransMux Layer

The TransMux Layer is a generic abstraction of the transport protocol stacks of existing delivery layers that may be used to transmit and store content complying with this Final Committee Draft of International Standard. The functionality of this layer is not in the scope of this specification, and only the interface to this layer is defined. It is called Stream Multiplex Interface (SMI) and may be embodied by the DMIF Application Interface (DAI) specified in Part 6 of this Final Committee Draft of International Standard. The SMI specifies an interface for streaming data only, while the scope of the DAI also covers signaling information. Similarly the TransMux Layer refers to data transport protocol stacks while the Delivery Layer specified in ISO/IEC FCD 14496-6 refers to signaling information as well. A wide variety of delivery mechanisms exists below this interface, with some indicated in Figure 1-1. These mechanisms serve for transmission as well as storage of streaming data, i.e., a file is considered a particular instance of a TransMux. For applications where the desired transport facility does not fully address the needs of an ISO/IEC FCD 14496 service, a simple multiplexing tool (FlexMux) is defined that provides low delay and low overhead.

1.5 Synchronization of Streams: Sync Layer

The Elementary Streams are the basic abstraction for any data source. Elementary Streams are conveyed as SL-packetized (Sync Layer-packetized) streams at the Stream Multiplex Interface. This packetized representation additionally provides timing and synchronization information, as well as fragmentation and random access information. The SL extracts this timing information to enable synchronized decoding and, subsequently, composition of the Elementary Stream data.

1.6 Compression Layer

The compression layer recovers data from its encoded format and performs the necessary operations to reconstruct the original information. The decoded information is then used by the terminal's composition, rendering and presentation subsystems. Access to the various Elementary Streams gained through object descriptors. An initial object descriptor needs to be made available through means not defined in this specification. The initial object descriptor points to one ore more initial Scene Description streams and the corresponding Object Descriptor streams.

An elementary stream may contain one of the following:

- object descriptors,
- audio or visual object data for a single object,
- scene description, or
- object content information.

The various Elementary Streams are described functionally below and specified in the normative clauses of this specification.

1.6.1 Object Descriptor Streams

The purpose of the object descriptor framework is to identify, describe and associate elementary streams with the various components of an audiovisual scene.

An object descriptor is a collection of one or more Elementary Stream descriptors that provide configuration and other information for the streams that relate to a single object (media object or scene description). Object Descriptors are themselves conveyed in elementary streams. Each object descriptor is assigned an identifying number (Object Descriptor ID), which is unique within the current session. This identifier is used to associate media 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. ES IDs are resolved to particular delivery channels at the TransMux layer. ES 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, for example to indicate dependence of an enhancement stream to its base stream in scalable audio or visual object representations, or the availability of the same speech content in various languages.

1.6.2 Scene Description Streams

Scene description addresses the organization of audiovisual objects in a scene, in terms of both spatial and temporal positioning. This information allows the composition and rendering of individual audiovisual objects after their respective decoders reconstruct them. This specification, however, does not mandate particular composition or rendering algorithms or architectures since they are implementation-dependent.

The scene description is represented using a parametric methodology (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 particular audio or visual objects (media nodes), whereas intermediate nodes perform grouping, transformation, and other operations (scene description nodes). The scene description can evolve over time by using scene description updates.

In order to allow active user involvement with the presented audiovisual information, this specification provides support for interactive operation. 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. This Final Committee Draft of International Standard, 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.

Local or client-side interactivity is provided via the routes and sensors mechanism of BIFS. Such an interactive environment does not need an upstream channel. This Final Committee Draft of International Standard also provides means for client-server interactive sessions with the ability to set up upchannel elementary streams.

1.6.3 Media Streams

The coded representations of audio and visual information are described in Parts 2 and 3, respectively, of this Final Committee Draft of International Standard. The reconstructed media objects are made available to the composition process for potential use during scene rendering.

1.6.4 Object Content Information Streams

An Object Content Information (OCI) stream carries descriptive information about audiovisual objects. The stream is organized in a sequence of small, synchronized entities called events that contain information descriptors. The main content descriptors are: content classification descriptors, keyword descriptors, rating descriptors, language descriptors, textual descriptors, and descriptors about the creation of the content. These streams can be associated to other media objects with the mechanisms provided by the Object Descriptor. When Object Content Information is not time variant, (and therefore does not need to be carried in an elementary stream by itself), it can be directly included in the related ES Descriptor(s).

1.6.5 Upchannel Streams

Downchannel elementary streams may require upchannel information to be transmitted from the receiver to the sender (e.g., to allow for client-server interactivity). An Elementary Stream flowing from receiver to sender is treated the same way as any downstream Elementary Stream as described in Figure 1-1. 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 Part 2 of this Final Committee Draft of International Standard.

2. Normative References

The following ITU-T Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Final Committee Draft of International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Final Committee Draft of International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau maintains a list of currently valid ITU-T Recommendations.

None cited.

3. Additional References

[1] ISO/IEC International Standard 13818-1 (MPEG-2 Systems), 1994.

[2] ISO/IEC 14772-1 International Standard, Virtual Reality Modeling Language (VRML), 1997.

[3] ISO 639, Code for the representation of names of languages, 1988.

[4] ISO 3166-1, Codes for the representation of names of countries and their subdivisions – Part 1: Country codes, 1997.

[5] The Unicode Standard, Version 2.0, 1996.

4. Definitions

Access Unit (AU): An individually accessible portion of the coded representation of an audiovisual object within an *elementary stream*. See also Subclause 7.2.3.

Alpha Map : The representation of the transparency parameters associated to a texture map.

Audiovisual Scene (AV Scene): A set of *media objects* together with *scene description* information that defines their spatial and temporal attributes including behavior resulting from object and user interaction.

Buffer Model: A model that defines how a terminal complying with this specification manages the buffer resources that are needed to decode a session.

Byte Aligned: A position in a coded bit stream with a distance of a multiple of 8-bits from the first bit in the stream.

Clock Reference: A special *timestamp* that conveys a reading of a *time base*.

Composition: The process of applying scene description information in order to identify the spatio-temporal attributes of media objects.

Composition Memory (CM): A random access memory that contains *composition units*. See also Subclause 7.2.9.

Composition Time Stamp (CTS): An indication of the nominal composition time of a *composition unit*. See also Subclause 7.3.5.

Composition Unit (CU): An individually accessible portion of the output that a *media object decoder* produces from *access units*. See also Subclause 7.2.8.

Compression Layer: The layer of an ISO/IEC FCD 14496 system that translates between the coded representation of an *elementary stream* and its decoded representation. It incorporates the *media object decoders*.

Decoding buffer (DB): A buffer at the input of a *media object decoder* that contains *access units* . See also Subclause 7.2.4.

Decoder configuration: The configuration of a *media object decoder* for processing its *elementary stream* data by using information contained in its *elementary stream descriptor*.

Decoding Time Stamp (DTS): An indication of the nominal decoding time of an *access unit*. See also Subclause 7.3.4.

Descriptor: A data structure that is used to describe particular aspects of an *elementary stream* or a coded *media object*.

Elementary Stream (ES): A consecutive flow of data from a single source entity to a single destination entity on the *compression layer*. See also Subclause 7.2.5.

Elementary Stream Descriptor: A structure contained in *object descriptors* that describes the encoding format, initialization information, transport channel identification, and other descriptive information about the content carried in an *elementary stream*. See also 9.3.3.

Elementary Stream Interface (ESI): An interface modeling the exchange of *elementary stream* data and associated control information between the *compression layer* and the *sync layer*. See also Subclause 7.2.6.

FlexMux Channel (FMC): A label to differentiate between data belonging to different constituent streams within one FlexMux Stream. A sequence of data within a *FlexMux stream* that corresponds to one single *SL-packetized stream*.

FlexMux Packet: The smallest data entity managed by the FlexMux tool consisting of a header and a payload.

FlexMux Stream: A sequence of *FlexMux Packets* with data from one or more *SL-packetized streams* that are each identified by their own *FlexMux channel.*.

FlexMux tool: A tool that allows the interleaving of data from multiple data streams.

Graphics Combination Profile: A *combination profile* that describes the required capabilities of a *terminal* for processing graphical *media objects*.

Inter: A mode for coding parameters that uses previously coded parameters to construct a prediction.

Intra: A mode for coding parameters that does not make reference to previously coded parameters to perform the encoding.

Initial Object Descriptor: A special *object descriptor* that allows the receiving terminal to gain access to portions of content encoded according to this specification.

Intellectual Property Identification (IPI): A unique identification of one or more *elementary streams* corresponding to parts of one or more *media objects*.

Media Object: A representation of a natural or synthetic object that can be manifested aurally and/or visually. Each object is associated with zero or more *elementary streams* using one ore more *object descriptors.*

Media Object Decoder: An entity that translates between the coded representation of an *elementary stream* and its decoded representation. See also Subclause 7.2.7.

Native BIFS Node: A BIFS node which is introduced and specified within this Final Committee Draft of International Standard as opposed to non-native BIFS node, which is a node referenced from ISO/IEC 14772-1.

Object Clock Reference (OCR): A *clock reference* that is used by a *media object decoder* to recover the encoder's *time base*. See also Subclause 7.3.3.

Object Content Information (OCI): Additional information about content conveyed through one or more *elementary streams*. It is either attached to to individual *elementary stream descriptors* or conveyed itself as an *elementary stream*.

Object Descriptor (OD): A descriptor that associates one or more *elementary streams* by means of their *elementary stream descriptors* and defines their logical dependencies.

Object Descriptor Message: A message that identifies the action to be taken on a list of *object descriptors* or object descriptor IDs, e.g., update or remove.

Object Descriptor Stream: An elementary stream that conveys *object descriptors* encapsulated in *object descriptor messages*.

Object Time Base (OTB): A *time base* valid for a given object, and hence for its *media object decoder*. The OTB is conveyed to the media object decoder via *object clock references*. All *timestamps* relating to this object's decoding process refer to this *time base*. See also Subclause 7.3.2.

Parametric Audio Decoder: A set of tools for representing and decoding audio (speech) signals coded at bit rates between 2 Kbps and 6 Kbps, according to Part 3 of this Final Committee Draft of International Standard.

Quality of Service (QoS): The performance that an *elementary stream* requests from the delivery channel through which it is transported, characterized by a set of parameters (e.g., bit rate, delay jitter, bit error rate).

Random Access: The process of beginning to read and decode a coded representation at an arbitrary point.

Reference Point: A location in the data or control flow of a system that has some defined characteristics.

Rendering: The action of transforming a scene description and its media objects from a common representation space to a specific presentation device (i.e., speakers and a viewing window).

Rendering Area: The portion of the display device's screen into which the scene description and its media objects are to be rendered.

Scene Description: Information that describes the spatio-temporal positioning of *media objects* as well as their behavior resulting from object and user interactions.

Scene Description Profile: A *profile* that defines the permissible set of *scene description* elements that may be used in a *scene description stream*.

Scene Description Stream: An elementary stream that conveys BIFS *scene description* information as specified in Subclause 9.2.12.

Session: The (possibly interactive) communication of the coded representation of an *audiovisual scene* between two *terminals*. A unidirectional session corresponds to a single program in a broadcast application.

SL-Packetized Stream (SPS): A sequence of SL-Packets that encapsulate one elementary stream. See also Subclause 7.2.2.

Stream Multiplex Interface (SMI): An interface modeling the exchange of *SL-packetized stream* data and associated control information between the *sync layer* and the *TransMux layer*. See also Subclause 7.2.1.

Kommentar: Replace TransMux by delivery ?

Structured Audio: A method of describing synthetic sound effects and music. See Part 3 of this Final Committee Draft of International Standard.

Sync Layer (SL): A layer to adapt *elementary stream* data for communication across the Stream Multiplex Interface, providing timing and synchronization information, as well as fragmentation and random access information. The Sync Layer syntax is configurable and can also be empty.

Sync Layer Configuration: A configuration of the *sync layer* syntax for a particular *elementary stream* using information contained in its *elementary stream descriptor*.

Sync Layer Packet (SL-Packet): The smallest data entity managed by the *sync layer* consisting of a configurable header and a payload. The payload may consist of one complete *access unit* or a partial *access unit*.

Syntactic Description Language (SDL): A language defined by this specification that allows the description of a bitstream's syntax.

Systems Decoder Model (SDM): A model that provides an abstract view of the behavior of a terminal compliant to this specification. It consists of the *Buffering Model,* and the *Timing Model*.

System Time Base (STB): The *time base* of the *terminal*. Its resolution is implementation-dependent. All operations in the *terminal* are performed according to this *time base*. See also Subclause 7.3.1.

Terminal: A system that receives and presents the coded representation of an interactive audiovisual scene as defined by this specification. It can be a standalone system, or part of an application system that supports presentation of content complying with this specification.

Time Base: The notion of a clock; it is equivalent to a counter that is periodically incremented.

Timing Model: A 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 terminal.

Time Stamp: An indication of a particular time instant relative to a *time base.*

TransMux: A generic abstraction for delivery mechanisms (computer networks, etc.) able to store or transmit a number of multiplexed *elementary streams* or *FlexMux streams*. This specification does not specify a TransMux layer.

Universal Resource Locator: A unique identification of the location of an *elementary stream* or an *object descriptor*.

5. Abbreviations and Symbols

The following symbols and abbreviations are used in this specification.

AU	Access Unit
AV	audiovisual
BIFS	Binary Format for Scene
CM	Composition Memory
CTS	Composition Time Stamp
CU	Composition Unit
DAI	DMIF Application Interface (see Part 6 of this Final Committee Draft of International Standard)
DB	Decoding Buffer
DTS	Decoding Time Stamp
ES	Elementary Stream
ESI	Elementary Stream Interface
ESID	Elementary Stream Identifier
FAP	Facial Animation Parameters
FAPU	FAP Units
FDP	Facial Definition Parameters
FIG	FAP Interpolation Graph
FIT	FAP Interpolation Table
FMC	FlexMux Channel
FMOD	The floating point modulo (remainder) operator which returns the remainder of x/y such that: fmod(x/y) = x – k*y, where k is an integer sgn(fmod(x/y)) = sgn(x) abs(fmod(x/y)) < abs(y)
IP	Intellectual Property
IPI	Intellectual Property Identification
NCT	Node Coding Tables
NDT	Node Data Type
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
SAOL	Structure Audio Orchestra Language
SASL	Structured Audio Score Language
SDL	Syntactic Description Language
SDM	Systems Decoder Model
SL	Synchronization Layer
SL-Packet	Synchronization Layer Packet
SMI	Stream Multiplex Interface
SPS	SL-packetized Stream
STB	System Time Base
TTS	Text-To-Speech
URL	Universal Resource Locator
VOP	Video Object Plane
VRML	Virtual Reality Modelling Language

6. Conventions

6.1 Syntax Description

For the purpose of unambiguously defining the syntax of the various bitstream components defined by the normative parts of this Final Committee Draft of International Standard 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 should be placed in a serialized bitstream. The definition of the language is provided in Subclause 1

7. Systems Decoder Model

7.1 Introduction

The purpose of the Systems Decoder Model (SDM) is to provide an abstract view of the behavior of a terminal complying to this Final Committee Draft of International Standard. It can be used by the sender to predict how the receiver will behave in terms of buffer management and synchronization when reconstructing the compressed audiovisual information. 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 (Stream Multiplex Interface),
2. decoding buffers for compressed data for each elementary stream,
3. the behavior of media object decoders, and
4. composition memory for decompressed data for each media object and the output behavior towards the compositor.

These elements are depicted in Figure 7-1. Each elementary stream is attached to one single decoding buffer. More than one elementary stream may be connected to a single media object decoder (e.g., in a decoder of a scaleable object).

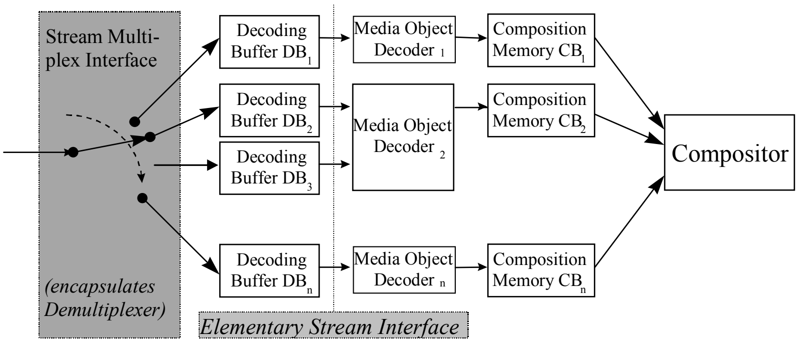


Figure 7-1: Systems Decoder Model

7.2 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 7-1.

7.2.1 Stream Multiplex Interface (SMI)

For the purposes of the Systems Decoder Model, the Stream Multiplex Interface encapsulates the demultiplexer, provides access to streaming data and fills up decoding buffers with this data. The streaming data received through the SMI consists of SL-packetized streams. The required properties of the SMI are described in Subclause 10.4. The SMI may be embodied by the DMIF Application Interface (DAI) specified in Part 6 of this Final Committee Draft of International Standard.

7.2.2 SL-Packetized Stream (SPS)

An SL-packetized stream consists of a sequence of packets, according to the syntax and semantics specified in Subclause 10.2.2, 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 enter the decoding buffers.

7.2.3 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 shall never refer to the same point in time. Any further structure 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 buffer. Access units with the necessary side information (e.g., time stamps) are taken from the decoding buffer through the Elementary Stream Interface.

Note: An ISO/IEC 14496-1 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 Subclause 10.2.4. This allows the encoder to dispatch partial AUs immediately as they are generated during the encoding process.

7.2.4 Decoding Buffer (DB)

The decoding buffer is a receiver buffer that contains access units. The Systems Buffering Model enables the sender to monitor the decoding buffer resources that are used during a session.

7.2.5 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 this specification. The integrity of an elementary stream is assumed to be preserved from end to end between two systems. Elementary streams are produced and consumed by Compression Layer entities (encoder, decoder).

7.2.6 Elementary Stream Interface (ESI)

The Elementary Stream Interface models the exchange of elementary stream data and associated control information between the Compression Layer and the Sync Layer. At the receiving terminal the ESI is located at the output of the decoding buffer. The ESI is specified in Subclause 10.3.

7.2.7 Media Object Decoder

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

7.2.8 Composition Units (CU)

Media object decoders produce composition units from access units. An access unit corresponds to an integer number of composition units. Composition units reside in composition memory.

7.2.9 Composition Memory (CM)

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

7.2.10 Compositor

The compositor takes composition units out of the composition memory and either composes and presents them or skips them. The compositor is not specified in this Final Committee Draft of International Standard, as the details of this operation are not relevant within the context of the System Decoder Model. Subclause 7.3.5 defines which composition unit is available to the Compositor at any instant of time.

7.3 Timing Model Specification

The timing model relies on clock references and time stamps to synchronize media objects 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 instant at which an event should take place in relation to a known clock. These time events are attached to access units and composition units. The semantics of the timing model are defined in the subsequent subclauses. The syntax for conveying timing information is specified in Subclause 10.2.4.

Note: This model is designed for rate-controlled ("push") applications.

7.3.1 System Time Base (STB)

The System Time Base (STB) defines the receiving terminal's notion of time. The resolution of this 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 compliant receiver terminals operate on one single STB.

7.3.2 Object Time Base (OTB)

The Object Time Base (OTB) defines the notion of time for a given media object. The resolution of this OTB can be selected as required by the application or as defined by a profile. All time stamps that the encoder inserts in a coded media object data stream refer to this time base. The OTB of an object is known at the receiver either by means of information inserted in the media stream, as specified in Subclause 10.2.4, or by indication that its time base is slaved to a time base conveyed with another stream, as specified in Subclause 10.2.3.

Note: Elementary streams may be created for the sole purpose of conveying time base information.

Note: The receiving terminal's System Time Base need not be locked to any of the available Object Time Bases.

7.3.3 Object Clock Reference (OCR)

A special kind of time stamps, Object Clock References (OCR), is used to convey the OTB to the media object decoder. The value of the OCR corresponds to the value of the OTB at the time the transmitting terminal generates the Object Clock Reference time stamp. OCR time stamps are placed in the SL packet header as described in Subclause 10.2.4. The receiving terminal shall extract and evaluate the OCR when its first byte enters its decoding buffer.

7.3.4 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 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 Subclause 10.2.3). Otherwise it is conveyed by a decoding time stamp (DTS) whose syntax is defined in Subclause 10.2.4.

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.

7.3.5 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. However, the CU is already available in the composition memory for use by the decoder (e.g. prediction) at the time indicated by DTS of the associated AU, since the SDM assumes instantaneous decoding.

This point in time is implicitly known, if the (constant) temporal distance between successive composition units is indicated in the setup of the elementary stream. Otherwise it is conveyed by a composition time stamp (CTS) whose syntax is defined in Subclause 10.2.4.

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 media object (i.e., when its object descriptor is removed).

7.3.6 Occurrence and Precision of Timing Information in Elementary Streams

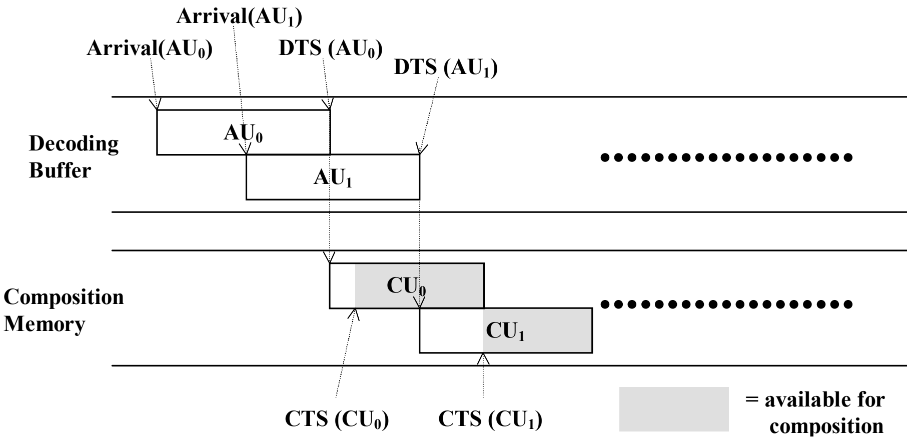
The frequency at which DTS, CTS and OCR values are to be inserted in the bitstream as well as the precision, jitter and drift are application and profile dependent.

7.3.7 Time Stamps for Dependent Elementary Streams

A media object may be represented in a scaleable manner by multiple elementary streams. 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.

7.3.8 Example

The example below illustrates the arrival of two access units at the Systems Decoder. Due to the constant delay assumption of the model (see Subclause 7.4.2 below), the arrival times correspond to the point in time when the transmitter has sent the respective AUs. The transmitter must select this point in time so that the Decoding Buffer never overflows or underflows. At DTS an AU is instantaneously decoded 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.



7.4 Buffer Model Specification

7.4.1 Elementary Decoder Model

The following simplified model is assumed for the purpose of specifying the buffer model. Each elementary stream is regarded separately.

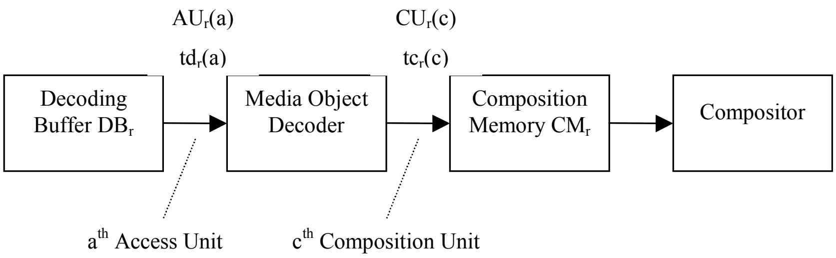


Figure 7-2: Flow diagram for the Systems Decoder Model

Legend:

r	Index to the different elementary streams.
DB_r	Decoding buffer for elementary stream r.
CM_r	Composition memory for elementary stream r.
a	Index to access units of one elementary stream.
c	Index to composition units of one elementary stream.
$AU_r(a)$	The a^{th} access unit in elementary stream r. $AU_r(a)$ is indexed in decoding order.
$td_r(a)$	The decoding time, measured in seconds, of the a^{th} access unit in the elementary stream ‘r’.
$CU_r(c)$	The c^{th} composition unit in elementary stream ‘r’. $CU_r(c)$ is indexed in composition order. $CU_r(c)$ results from decoding $AU_r(a)$. There may be several composition units resulting from decoding one access unit
$tc_r(c)$	The composition time, measured in seconds, of the c^{th} composition unit in the elementary stream ‘r’.

7.4.2 Assumptions

7.4.2.1 Constant end-to-end delay

Media objects being presented and transmitted in real time have a timing model in which the end-to-end delay from the encoder input to the decoder output is a constant. This delay is the sum of encoding, encoder buffering, multiplexing, communication or storage, demultiplexing, decoder buffering and decoding delays.

Note that the decoder 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.

7.4.2.2 Demultiplexer

The end-to-end delay between multiplexer output and demultiplexer input is constant.

7.4.2.3 Decoding Buffer

The needed decoding buffer size is known by the sender and conveyed to the receiver as specified in Subclause 8.3.4.

The size of the decoding buffer is measured in bytes.

Decoding buffers are filled at the rate given by the maximum bit rate for this elementary stream if data is available from the demultiplexer, and with a zero rate otherwise. The maximum bit rate is conveyed in the decoder configuration during set up of each elementary stream (see Subclause 8.3.4).

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

7.4.2.4 Decoder

The decoding time is assumed to be zero for the purposes of the Systems Decoder Model.

7.4.2.5 Composition Memory

The mapping of an AU to one or more CUs is known implicitly (by the decoder) to both the sender and the receiver.

7.4.2.6 Compositor

The composition time is assumed to be zero for the purposes of the Systems Decoder Model.

7.4.3 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 transmitted, flow control by suitable signaling may be established to request access units at the time they are needed at the receiver. The mechanisms for doing so are application-dependent, and are not specified in this Final Committee Draft of International Standard.

The behavior of the various SDM elements is modeled as follows:

- The sender signals the required buffer resources to the receiver before starting the transmission. This is done as specified in Subclause 8.3.4 either explicitly by requesting buffer sizes for individual elementary streams or implicitly by specification of a profile. The buffer size is measured in bytes.
- The sender models the buffer behavior by making the following assumptions :
- ▪ The decoding buffer is filled at the maximum bitrate for this elementary stream if data is available.
- ▪ At DTS, an AU is instantaneously decoded and removed from the DB.
- ▪ At DTS, a known amount of CUs corresponding to the AU are put in the composition memory,
- ▪ The current CU is available to the compositor between its composition time and the composition time of the subsequent CU. If a subsequent CU does not exist, the CU becomes unavailable at the end of lifetime of its media object.

With these model assumptions the sender may freely use the space in the buffers. For example, it may transfer data for several AUs of a non-real time stream to the receiver, and pre-store them in the DB long before they have to be decoded (assuming sufficient space is available). Afterwards, the full channel 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 to contain decoded P-frames which are needed by the video decoder for the decoding of intermediate B-frames before the arrival of the CTS of the latest P-frame.

8. Object Descriptors

8.1 Introduction

The scene description (specified in Clause 9) and the elementary streams that convey audio or visual objects – as well as the scene description itself – are the basic building blocks of the architecture of this Final Committee Draft of International Standard. However, the scene description does not directly refer to elementary streams when specifying a media object, but uses the concept of object descriptors. This provides an indirect mechanism that facilitates the separation between scene structure, media data, and transport facilities used, so that changes to any one of these can be performed without affecting the others.

The purpose of the object descriptor framework is to identify and poperly associate elementary streams to each other and to media objects used in the scene description. The scene description declares the spatio-temporal relationship of the available media objects, while object descriptors identify and describe the elementary stream resources that provide the content. Those media objects that necessitate elementary stream data point to an object descriptor by means of a numeric identifier, an *objectDescriptorID*.

Each object descriptor is itself a collection of descriptors that describe the elementary stream(s) comprising a single media object. Such streams may contain compressed media data as well as metadata that may potentially be time variant (Object Content Information). Dependencies between streams may be signaled to indicate, for example, a scaleable content representation. Furthermore, multiple alternative streams that convey the same content, e.g., in multiple qualities or different languages, may be associated to a single media object.

An ES_Descriptor identifies a single stream with a numeric identifier, ES_ID, and an optional URL pointing to a remote source for the stream. Each ES_Descriptor contains the information necessary to initiate and configure the decoding process for the stream, as well as intellectual property identification. Optionally, additional information can be associated to an elementary stream, most notably quality of service requirements for its transmission or a language indication.

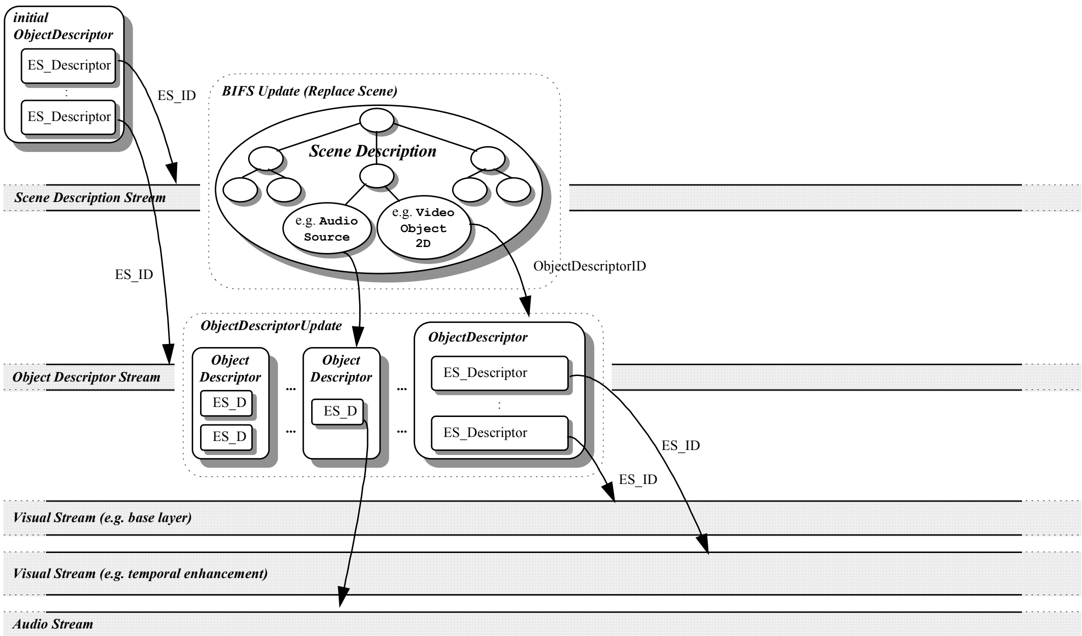


Figure 8-1: The Object Descriptor Framework

Access to content complying with this Final Committee Draft of International Standard is gained through an initial object descriptor that needs to be made available through means not defined in this specification (session setup). The initial object descriptor points to the scene description stream and the corresponding set of object descriptors that are themselves conveyed in an elementary stream. The access scenario is outlined in Figure 8-1.

This subclause specifies the process of accessing content complying with this Final Committee Draft of International Standard using object descriptors, as well as their syntax and semantics and the method to convey the object descriptors. In particular:

- Subclause 8.2 specifies the object descriptor stream and the syntax and semantics of the message set that allows the update or removal of object descriptor components.
- Subclause 8.3 specifies the syntax and semantics of the object descriptor and its sub-descriptors.
- Subclause 8.4 specifies object descriptor usage, including the procedure to access content through object descriptors.

8.2 Object Descriptor Stream

8.2.1 Structure of the Object Descriptor Stream

Similar to the scene description, object descriptors are transported in a dedicated elementary stream that allows to convey, update and remove complete object descriptors and their primary component, the ES_Descriptors, anytime during the course of a session. Some semantic restrictions exist, and are detailed below. The update mechanism allows to advertise new elementary streams for a media 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. Note that object descriptor updates need not be co-incident in time with the addition or removal of media objects in the scene description that refer to such an object descriptor. However, media objects referenced in the scene description must have a valid object descriptor at all times that they are present in the scene.

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

8.2.2 Principles of Syntax Specification and Parsing

The syntax of each message defined in Subclause 8.2.7 and each descriptor defined in Subclause 8.3 constitutes a self-describing class, identified by a unique class tag. The class tag values for all classes are defined in Subclause 8.3.12. Normative sequences of classes related to the object descriptor framework are defined in the remainder of this clause.

In order to facilitate future compatible extensions, interspersed classes with unknown class tag values are permissible and shall be ignored.

8.2.3 OD Messages

Object descriptors and their components shall be conveyed as part of one of the OD messages specified in Subclause 8.2.7. The messages describe the action to be taken on the components conveyed with the message, specifically 'update' or 'remove'. Each message affects one or more object descriptors or ES_Descriptors. Each OD message shall have an associated point in time at which it becomes valid.

8.2.4 Access Unit Definition

All OD messages that refer to the same instant in time shall constitute a single access unit. Access units in object descriptor elementary streams shall be labeled 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 (Subclause 10.2.4). The composition time indicates the point in time when an OD access unit becomes valid. Decoding and composition time for an OD access unit shall always have the same value.

An access unit does not necessarily convey or update all object descriptors that are currently required. 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'.

8.2.5 Time Base for Object Descriptor Streams

As with any elementary stream, the object descriptor stream has an associated time based as specified in Subclause 7.3.2. The syntax to convey time bases to the receiver is specified in Subclause 10.2. It is also possible to incidate than an object descriptor stream uses the time base of another elementary stream (see Subclause 8.3.6). All time stamps refer to this time base.

8.2.6 Implicit Length of Descriptor Lists

The content of some OD messages consists of a list of object descriptors or ES_Descriptors. The length of this list is determined implicitly. The message is complete if either the next OD message class is encountered or if the current access unit is terminated.

8.2.7 OD Message Syntax and Semantics

8.2.7.1 ObjectDescriptorUpdate

8.2.7.1.1 Syntax

```
aligned(8) class ObjectDescriptorUpdate
      : bit(8) tag=ObjectDescrUpdateTag {
   bit(8) length;
   ObjectDescriptor OD[1 .. 255];
}
```

8.2.7.1.2 Semantics

The `ObjectDescriptorUpdate` class conveys a list of new or updated ObjectDescriptors. The components of an already existing ObjectDescriptor shall not be changed by an update, but an `ObjectDescriptorUpdate` may remove or add ES_Descriptors as components of the related object descriptor.

To change the characteristics of an elementary stream it is required to remove its ES_Descriptor and subsequently convey the changed ES_Descriptor.

`length` – length of the remainder of this message in bytes excluding trailing embedded descriptors. Length shall be equal to zero.

`OD[]` – an array of `ObjectDescriptors` as defined in Subclause 8.3. The array shall have any number of one up to 255 elements.

8.2.7.2 ObjectDescriptorRemove

8.2.7.2.1 Syntax

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

8.2.7.2.2 Semantics

The `ObjectDescriptorRemove` class renders unavailable a set of object descriptors. The media objects associated to these object descriptors shall have no reference any more to the elementary streams that have been listed in the removed object descriptors.

`length` – length of the remainder of this message in bytes. Length shall be greater than or equal to two (2).

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

8.2.7.3 ObjectDescriptorRemoveAll

8.2.7.3.1 Syntax

```
aligned(8) class ObjectDescriptorRemoveAll
      : bit(8) tag=ObjectDescrRemoveAllTag {
   bit(8) length;
}
```

8.2.7.3.2 Semantics

The `ObjectDescriptorRemoveAll` class renders unavailable all ObjectDescriptors in the name scope of this object descriptor stream (see Subclause 8.4.1.3). The media objects associated to these object descriptors shall have no reference any more to the elementary streams that have been listed in the removed object descriptors.

`length` – length of the remainder of this message in byte. It shall be equal to zero.

8.2.7.4 ES_DescriptorUpdate

8.2.7.4.1 Syntax

```
aligned(8) class ES_DescriptorUpdate : bit(8) tag=ES_DescrUpdateTag {
   bit(8) length;
   bit(10) objectDescriptorId;
   ES_Descriptor ESD[1 .. 30];
}
```

8.2.7.4.2 Semantics

The `ES_DescriptorUpdate` class adds or updates references to elementary streams in an `ObjectDescriptor`. Values of syntax elements of an updated ES_Descriptor shall remain unchanged.

To change the characteristics of an elementary stream it is required to remove its ES_Descriptor and subsequently convey the changed ES_Descriptor.

`length` – length of the remainder of this message in byte excluding trailing embedded descriptors.

`objectDescriptorID` - identifies the `ObjectDescriptor` for which `ES_Descriptors` are updated.

`ESD[]` – an array of `ES_Descriptors` as defined in Subclause 8.3.3. The array shall have any number of one up to 30 elements.

8.2.7.5 ES_DescriptorRemove

8.2.7.5.1 Syntax

```
aligned(8) class ES_DescriptorRemove : bit(8) ES_DescrRemoveTag {
   bit(8) length;
   bit(10) objectDescriptorId;
   bit(5) streamCount;
   const bit(1) reserved=1;
   bit(16) ES_ID[streamCount];
}
```

8.2.7.5.2 Semantics

The `ES_DescriptorRemove` class removes the reference to an elementary stream from an ObjectDescriptor and renders this stream unavailable for the associated media object.

`length` – length of the remainder of this message in bytes excluding trailing embedded descriptors.

`objectDescriptorID` - identifies the `ObjectDescriptor` from which `ES_Descriptors` are removed.

`streamCount` - indicates the number of `ES_Descriptors` to be removed.

ES_ID[streamCount] – an array of streamCount ES_IDs that label the ES_Descriptors to be removed from objectDescriptorID.

8.3 Syntax and Semantics of Object Descriptor Components

8.3.1 ObjectDescriptor

8.3.1.1 Syntax

```
aligned(8) class ObjectDescriptor : bit(8) tag=ObjectDescrTag {
   bit(8) length;
   bit(10) ObjectDescriptorID;
   bit(1) URL_Flag;
   const bit(5) reserved=0b1111.1;
   if (URL_Flag) {
      bit(8) URLstring[length-2];
   }
   ExtensionDescriptor extDescr[0 .. 255];
      if (!URL_Flag) {
         OCI_Descriptor ociDescr[0 .. 255];
      ES_Descriptor esDescr[1 .. 30];
   }
}
```

8.3.1.2 Semantics

The ObjectDescriptor consists of three different parts.

The first part uniquely labels the object descriptor within its name scope (see Subclause 8.4.1.3) by means of an objectDescriptorId. Media objects in the scene description use objectDescriptorID to refer to their object descriptor. An optional URLstring indicates that the actual object descriptor resides at a remote location.

The second 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.

The third part consists of a list of ES_Descriptors, each providing parameters for a single elementary stream that relates to the media object as well as an optional set of Object Content Information descriptors.

length – length of the remainder of this descriptor in byte excluding trailing embedded descriptors.

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.

URLstring[] – A string with a URL that shall point to another ObjectDescriptor.

extDescr[] – An array of ExtensionDescriptors as defined in Subclause 8.3.10. The array shall have any number of zero up to 255 elements.

ociDescr[] – An array of OCI_Descriptors, as defined in Subclause 13.2.3, that relate to the media object described by this object descriptor. The array shall have any number of zero up to 255 elements.

ESD[] – An array of ES_Descriptors as defined in Subclause 8.3.3. The array shall have any number of one up to 30 elements.

8.3.2 InitialObjectDescriptor

8.3.2.1 Syntax

```
aligned(8) class InitialObjectDescriptor
      : bit(8) tag=InitialObjectDescrTag {
   bit(8) length;
```

```
   bit(10) ObjectDescriptorID;
   bit(1) URL_Flag;
   bit(1) includeInlineProfilesFlag;
   const bit(4) reserved=0b1111.1;
   if (URL_Flag) {
      bit(8) URLstring[length-2];
   } else {
      bit(8) sceneProfile;
      bit(8) audioProfile;
      bit(8) visualProfile;
      bit(8) graphicsProfile;
   }
   ExtensionDescriptor extDescr[0 .. 255];
   if (!URL_Flag) {
      OCI_Descriptor ociDescr[0 .. 255];
      ES_Descriptor ESD[1 .. 30];
   }
}
```

8.3.2.2 Semantics

The `InitialObjectDescriptor` is a variation of the `ObjectDescriptor` specified in the previous subclause that shall be used to gain initial access to content complying with this Final Committee Draft of International Standard (see Subclause 8.4).

`length` – length of the remainder of this descriptor in bytes excluding trailing embedded descriptors.

`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`.

`includeInlineProfilesFlag` – 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.

`URLstring[]` – A string with a URL that shall point to another `InitialObjectDescriptor`.

`sceneProfile` – an indication of the scene description profile required to process the content associated with this `InitialObjectDescriptor`.

Table 8-1: sceneProfile Values

Value	`sceneProfile` **Description**
0x00	Reserved for ISO use
0x01	Systems 14496 1 XXXX profile
0x??-0x7F	reserved for ISO use
0x80-0xFD	user private
0xFE	no scene description profile specified
0xFF	no scene description capability required

`audioProfile` – an indication of the audio profile required to process the content associated with this `InitialObjectDescriptor`.

Table 8-2: audioProfile Values

Value	`audioProfile` **Description**
0x00	Reserved for ISO use
0x01	Systems 14496 3 XXXX profile
0x??-0x7F	reserved for ISO use
0x80-0xFD	user private
0xFE	no audio profile specified

0xFF	no audio capability required

`visualProfile` – an indication of the visual profile required to process the content associated with this `InitialObjectDescriptor`.

Table 8-3: visualProfile Values

Value	`visualProfile` **Description**
0x00	Reserved for ISO use
0x01	Systems 14496 2 XXXX profile
0x??-0x7F	reserved for ISO use
0x80-0xFD	user private
0xFE	no visual profile specified
0xFF	no visual capability required

`graphicsProfile` – an indication of the graphics profile required to process the content associated with this `InitialObjectDescriptor`.

Table 8-4: graphicsProfile Values

Value	`graphicsProfile` **Description**
0x00	Reserved for ISO use
0x01	Systems 14496 1 XXXX profile
0x??-0x7F	reserved for ISO use
0x80-0xFD	user private
0xFE	no graphics profile specified
0xFF	no graphics capability required

`extDescr[]` – An array of `ExtensionDescriptors` as defined in Subclause 8.3.10. The array shall have any number of zero up to 255 elements.

`ociDescr[]` – An array of OCI_Descriptors as defined in Subclause 13.2.3 that relate to the set of media objects that are described by this initial object descriptor. The array shall have any number of zero up to 255 elements.

`ESD[]` – An array of `ES_Descriptors` as defined in Subclause 8.3.3. The array shall have any number of one up to 30 elements.

8.3.3 ES_Descriptor

8.3.3.1 Syntax

```
aligned(8) class ES_Descriptor : bit(8) tag=ES_DescrTag {
   bit(8)   length;
   bit(16)  ES_ID;
   bit(1)   streamDependenceFlag;
   bit(1)   URL_Flag;
   const bit(1) reserved=1;
   bit(5)   streamPriority;
   if (streamDependenceFlag)
      bit(16) dependsOn_ES_ID;
   if (URL_Flag)
      bit(8) URLstring[length-3-(streamDependencFlag*2)];
   ExtensionDescriptor     extDescr[0 .. 255];
   LanguageDescriptor      langDescr[0 .. 1];
   DecoderConfigDescriptor decConfigDescr;
   SLConfigDescriptor      slConfigDescr;
   IPI_DescPointer         ipiPtr[0 .. 1];
```

```
   IP_IdentificationDataSet ipIDS[0 .. 1];
   QoS_Descriptor           qosDescr[0 .. 1];
}
```

8.3.3.2 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 Subclause 8.4.1.3), a mechanism to group elementary streams within this `ObjectDescriptor` and an optional URL string. Grouping of elementary streams and usage of URLs are specified in Subclause 8.4.

The second 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.

The third part consists of the `DecoderConfigDescriptor`, `SLConfigDescriptor`, `IPI_Descriptor` and `QoS_Descriptor` structures which convey the parameters and requirements of the elementary stream.

`length` – length of the remainder of this descriptor in byte excluding trailing embedded descriptors.

`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.

`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 this `ObjectDescriptor`.

`URLstring[]` – contains a 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`.

`extDescr[]` – is an array of `ExtensionDescriptor` structures as specified in Subclause 8.3.10.

`langDescr[]` – is an array of zero or one `LanguageDescriptor` structures as specified in Subclause 13.2.3.6. It indicates the language attributed to this elementary stream.

`decConfigDescr` – is a `DecoderConfigDescriptor` as specified in Subclause 8.3.4.

`slConfigDescr` – is an `SLConfigDescriptor` as specified in Subclause 8.3.6.

`ipiPtr[]` – is an array of zero or one `IPI_DescPointer` as specified in Subclause 8.3.8.

`ipIDS[]` – is an array of zero or one `IP_IdentificationDataSet` as specified in Subclause 8.3.7.

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

`qosDescr[]` – is an array of zero or one `QoS_Descriptor` as specified in Subclause 8.3.9.

8.3.4 DecoderConfigDescriptor

8.3.4.1 Syntax

```
aligned(8) class DecoderConfigDescriptor
      : bit(8) tag=DecoderConfigDescrTag {
   bit(8) length;
   bit(8) objectProfileIndication;
   bit(6) streamType;
   bit(1) upStream;
   const bit(1) reserved=1;
   bit(24) bufferSizeDB;
   bit(32) maxBitrate;
```

```
   bit(32) avgBitrate;
   DecoderSpecificInfo decSpecificInfo[];
}
```

8.3.4.2 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 receiver 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.

`length` – length of the remainder of this descriptor in bytes excluding trailing embedded descriptors.

`objectProfileIndication` – an indication of the object profile, or scene description profile if `streamType`=sceneDescriptionStream, that needs to be supported by the decoder for this elementary stream as per this table. For `streamType`=ObjectDescriptorStream the value `objectProfileIndication`=’no profile specified’ shall be used.

Table 8-5: objectProfileIndication Values

Value	**`objectProfileIndication` Description**
0x00	Reserved for ISO use
0x01	Systems 14496 1 Simple Scene Description
0x02	Systems 14496 1 2D Scene Description
0x03	Systems 14496 1 VRML Scene Description
0x04	Systems 14496 1 Audio Scene Description
0x05	Systems 14496-1 Complete Scene Description
0x06-0x1F	reserved for ISO use
0x20	Visual 14496-2 simple profile
0x21	Visual 14496-2 core profile
0x22	Visual 14496-2 main profile
0x23	Visual 14496-2 simple scalable profile
0x24	Visual 14496-2 12-Bit
0x25	Visual 14496 -2 Basic Anim. 2D Texture
0x26	Visual 14496-2 Anim. 2D Mesh
0x27	Visual 14496-2 Simple Face
0x28	Visual 14496-2 Simple Scalable Texture
0x29	Visual 14496-2 Core Scalable Texture
0x2A-0x3F	reserved for ISO use
0x40	Audio 14496-3 AAC Main
0x41	Audio 14496-3 AAC LC
0x42	Audio 14496-3 T/F
0x43	Audio 14496-3 T/F Main scalable
0x44	Audio 14496-3 T/F LC scalable
0x45	Audio 14496-3 Twin VQ core
0x46	Audio 14496-3 CELP
0x47	Audio 14496-3 HVXC
0x48	Audio 14496-3 HILN
0x49	Audio 14496-3 TTSI
0x4A	Audio 14496-3 Main Synthetic
0x4B	Audio 14496-3 Wavetable Synthesis
0x4C-0x5F	reserved for ISO use
0x60	Visual 13818-2 Simple Profile
0x61	Visual 13818-2 Main Profile
0x62	Visual 13818-2 SNR Profile
0x63	Visual 13818-2 Spatial Profile
0x64	Visual 13818-2 High Profile

0x65	Visual 13818-2 422 Profile
0x66	Audio 13818-7
0x67	Audio 13818-3
0x68	Visual 11172-2
0x69	Audio 11172-3
0x6A - 0xBF	reserved for ISO use
0xC0 - 0xFE	user private
0xFF	no profile specified

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

Table 8-6: streamType Values

`streamType` value	**stream type description**
0x00	reserved for ISO use
0x01	ObjectDescriptorStream (Subclause 8.2)
0x02	ClockReferenceStream (Subclause 10.2.5)
0x03	SceneDescriptionStream (Subclause 9.2.12)
0x04	VisualStream
0x05	AudioStream
0x06	MPEG7Stream
0x07-0x09	reserved for ISO use
0x0A	ObjectContentInfoStream (Subclause 13.2)
0x0B - 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 of this elementary stream in any time window of one second duration.

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

`decSpecificInfo[]` – an array of decoder specific information as specified in Subclause 8.3.5.

8.3.5 DecoderSpecificInfo

8.3.5.1 Syntax

```
abstract aligned(8) class DecoderSpecificInfo : bit(8) tag=0
{
}

aligned(8) class DecoderSpecificInfoShort extends DecoderSpecificInfo
      : bit(8) tag=DecSpecificInfoShortTag
{
   bit(8) length;
   bit(8) specificInfo[length];
}

aligned(8) class DecoderSpecificInfoLarge extends DecoderSpecificInfo
      : bit(8) tag=DecSpecificInfoLargeTag
{
   bit(32) length;
   bit(8) specificInfo[length];
}
```

8.3.5.2 Semantics

The decoder specific information constitutes an opaque container with information for a specific media decoder. Depending on the required amount of data, two classes with a maximum of 255 and 2^{32}-1 bytes of data are provided. The existence and semantics of decoder specific information depends on the values of `DecoderConfigDescriptor.streamType` and `DecoderConfigDescriptor.objectProfileIndication`.

For values of `DecoderConfigDescriptor.objectProfileIndication` that refer to streams complying with Part 2 of this Final Committee Draft of International Standard the semantics of decoder specific information are defined in Annex L of that part. For values of `DecoderConfigDescriptor.objectProfile Indication` that refer to Part 3 of this Final Committee Draft of International Standard the semantics of decoder specific information are defined in Clause 3.1.1 of that part. For values of `DecoderConfigDescriptor.objectProfileIndication` that refer to scene description streams the semantics of decoder specific information is defined in Subclause 9.4.1.1.

`length` – length of the remainder of this descriptor in byte.

`specificInfo[length]` – an array of `length` byte of decoder specific information.

8.3.6 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 Subclause 10.2.3.

8.3.7 IP Identification Data Set

8.3.7.1 Syntax

```
aligned(8) class IP_IdentificationDataSet
      : bit(8) tag=IP_IdentificationDataSetTag
{
   int         i;
   bit(8)      length;
   const bit(2)   compatibility=0;
   bit(1)      contentTypeFlag;
   bit(1)      contentIdentifierFlag;
   aligned(8) bit(8) supplementaryContentIdentifierCount;
   if (contentTypeFlag)
      bit(8)   contentType;
   if (contentIdentifierFlag) {
      bit(8)   contentIdentifierType;
      bit(8)   contentIdentifierLength;
      bit(8)   contentIdentifier[contentIdentifierLength];
   }
   if (supplementaryContentIdentifierCount>0) {
      bit(24)  languageCode;
      for (i=0; i < supplementaryContentIdentifierCount; i++) {
         bit(8)    supplementaryContentIdentifierTitleLength;
                  bit(8) supplementaryContentIdentifierTitle[[i]]
                     [supplementaryContentIdentifierTitleLength];
         bit(8)    supplementaryContentIdentifierValueLength;
                  bit(8) supplementaryContentIdentifierValue[[i]]
                     [supplementaryContentIdentifierValueLength];
      }
   }
}
```

8.3.7.2 Semantics

The Intellectual Property Identification Data Set 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 `IP_IdentificationDataSet` may be associated to one elementary stream. The IPI information shall never be detached from the ES_Descriptor.

`length` – length of the remainder of this descriptor in bytes.

`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.

`supplementaryContentIdentifierCount` – since not all works follow a numbered identification scheme, non-standard schemes can be used (which can be alphanumerical or binary). The `supplementaryContentIdentifierCount` indicates how many of these "supplementary" data fields are following.

`contentType` – defines the type of content using one of the values specified in the the following table.

Table 8-7: 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 the following table.

Table 8-8: 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	

`contentIdentifierLength` – since the length of each of these identifiers can vary, a length indicator is needed to give the length in byte.

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

`language code` – This 24 bits field contains the ISO 639 three character language code of the language of the following text fields.

`supplementaryContentIdentifierTitle` and `supplementaryContentIdentifierValue` - Each of these two entries give a title-and-value pair whenever a numeric content definition is not available. The length of the title (in bytes) is indicated by `supplementaryContentIdentifierTitleLength` (0 byte to 255 bytes) and the length of the `supplementaryContentIdentifierValue` is indicated by the `supplementaryContentIdentifierValueLength` (0 to 255 bytes).

8.3.8 IPI_DescPointer

8.3.8.1 Syntax

```
aligned(8) class IPI_DescPointer : bit(8) tag=IPI_DescPointerTag {
   bit(8) length;
   bit(16) IPI_ES_Id;
}
```

8.3.8.2 Semantics

The `IPI_DescPointer` class contains a reference to the elementary stream that includes the IP Identification Data Set(s) that are valid for this stream. This indirect reference mechanism allows to convey `IP_IdentificationDataSet` elements only in one elementary stream while making references to it from any `ES_Descriptor` that shares the same IP Identification Data.

`ES_Descriptors` for elementary streams that are intended to be accessible regardless of the availability of a referred stream shall explicitly include their IP Identification Data Set(s) instead of using an `IPI_DescPointer`.

`length` – length of the remainder of this descriptor in bytes.

`IPI_ES_Id` – the `ES_ID` of the elementary stream that 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.

8.3.9 QoS_Descriptor

8.3.9.1 Syntax

```
aligned(8) class QoS_Descriptor : bit(8) tag=QoS_DescrTag {
   int    i;
   bit(8) length;
   bit(8) predefined;
   if (predefined==0) {
      bit(8) QoS_QualifierCount;
      for (i=0; i<QoS_QualifierCount; i++) {
         bit(8) QoS_QualifierTag[[i]];
         bit(8) QoS_QualifierLength[[i]];
         bit(8) QoS_QualifierData[[i]][QoS_QualifierLength[[i]]];
      }
   }
}
```

8.3.9.2 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.

`length` – length of the remainder of this descriptor in byte.

`predefined` – a value different from zero indicates a predefined QoS profile according to the table below.

Table 8-9: Predefined QoS Profiles

predefined			
0x00	Custom		

0x01 - 0xff	Reserved		

`QoS_QualifierCount` – number of QoS metrics specified in the descriptor.

`QoS_QualifierTag[[i]]` – identifies the type of metric.

`QoS_QualifierLength[[i]]` – length of the following metric value.

`QoS_QualifierData[[i]][]` – the QoS metric value.

Table 8-10: List of QoS_QualifierTags

QoS_QualifierTag	Value	Description
Reserved	0x00	Reserved for ISO use
MAX_DELAY	0x01	Maximum end to end delay for the stream
PREF_MAX_DELAY	0x02	Preferred end to end delay for the stream
LOSS_PROB	0x03	Allowable loss probability of any single AU
MAX_GAP_LOSS	0x04	Maximum allowable number of consecutively lost AUs
Reserved	0x05-0x40	Reserved for ISO use
MAX_AU_SIZE	0x41	Maximum size of an AU
AVG_AU_SIZE	0x42	Average size of an AU
MAX_AU_RATE	0x43	Maximum arrival rate of AUs
Reserved	0x44-0x7f	Reserved for ISO use
User defined	0x80-0xff	User Private

Table 8-11: Length and units of QoS metrics

QoS_QualifierTag	Type	Unit
MAX_DELAY	Long	Microseconds
AVG_DELAY	Long	Microseconds
LOSS_PROB	Double	Fraction (0.00 – 1.00)
MAX_GAP_LOSS	Long	Integer number of access units (AU)
MAX_AU_SIZE	Long	Bytes
MAX_AU_RATE	Long	AUs/second
AVG_AU_SIZE	Double	Bytes

8.3.10 ExtensionDescriptor

8.3.10.1 Syntax

```
abstract aligned(8) class ExtensionDescriptor : bit(8) tag=0 {
}

abstract aligned(8) class ShortExtensionDescriptor
         extends ExtensionDescriptor
         : bit(8) tag=0
{
   bit(8) length;
   bit(8) descriptorData[length];
}

abstract aligned(8) class LongExtensionDescriptor
         extends ExtensionDescriptor
         : bit(8) tag=0
{
   bit(16) length;
   bit(8) descriptorData[length];
}
```

```
abstract aligned(8) class LargeExtensionDescriptor
         extends ExtensionDescriptor
         : bit(8) tag=0
{
   bit(32) length;
   bit(8) descriptorData[length];
}
```

8.3.10.2 Semantics

Additional descriptors may be defined using the syntax in the class definitions above. Depending on the value of the class tag, the maximum length of the descriptor can be either 2^8-1, 2^{16}-1 or 2^{32}-1 byte.

These descriptors may be ignored by a terminal that conforms to this specification. The available class tag values for extension descriptors allow ISO defined extensions as well as private extensions.

`length` – length of the remainder of this descriptor in byte.

`descriptorData[length]` – is an array of `length` data bytes.

8.3.11 RegistrationDescriptor

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

8.3.11.1 Syntax

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

8.3.11.2 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 this specification to unambiguously carry elementary streams with data whose format is not recognized by this specification. This provision will permit this specification 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 F, the benefits and responsibilities of all parties to the registration of private data format are outlined.

8.3.11.2.1 Implementation of a Registration Authority (RA)

ISO/IEC JTC1/SC29 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 clause. The selected organization shall serve as the Registration Authority. The so-named Registration Authority shall execute its duties in compliance with Annex H of the JTC1 directives. The registered private data formatIdentifier is hereafter referred to as the Registered Identifier (RID).

Upon selection of the Registration Authority, JTC1 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 this specification has been denied by the Registration Authority.

Annexes F and G to this Specification provide information on the procedures for registering a unique format identifier.

Kommentar: Make sure Annex numbers here are correct (not automagically produced).

8.3.12 Descriptor Tags

All classes specified in Subclauses 8.2 and 8.3 are identified by means of a class tag. The values of these class tags are defined in Table 8-12. The value of the descriptor tag determines whether the subsequent length field in the descriptor has a size of 8, 16 or 32 bit.

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.

Table 8-12: List of Descriptor Tags

Tag value	Tag name	Size of `length` field
0x00	Reserved for ISO use	8
0x01	ObjectDescrUpdateTag	8
0x02	ObjectDescrRemoveTag	8
0x03	ObjectDescrRemoveAllTag	8
0x04	ES_DescrUpdateTag	8
0x05	ES_DescrRemoveTag	8
0x06-0x1F	Reserved for ISO use for future message tags	8
0x20	ObjectDescrTag	8
0x21	InitialObjectDescrTag	8
0x22	ES_DescrTag	8
0x23	DecoderConfigDescrTag	8
0x24	DecSpecificInfoShortTag	8
0x25	SLConfigDescrTag	8
0x26	IP_IdentificationDataSetTag	8
0x27	IPI_DescPointerTag	8
0x28	QoS_DescrTag	8
0x29-0x3F	Reserved for ISO use	8
0x40	ContentClassificationDescrTag	8
0x41	KeyWordingDescrTag	8
0x42	RatingDescrTag	8
0x43	LanguageDescrTag	8
0x44	ShortTextualDescrTag	8
0x45	ContentCreatorNameDescrTag	8
0x46	ContentCreationDateDescrTag	8
0x47	OCICreatorNameDescrTag	8
0x48	OCICreationDateDescrTag	8
0x49-0x4F	Reserved for ISO use	8
0x50	RegistrationDescrTag	8
0x51-0x7F	reserved for ISO use as ShortExtensionDescriptor	8
0x80-0xAF	user private as ShortExtensionDescriptor	8
0xB0	ExpandedTextualDescrTag	16
0xB1-0xCF	reserved for ISO use as LongExtensionDescriptor	16
0xD0-0xDF	user private as LongExtensionDescriptor	16
0xE0	DecSpecificInfoLargeTag	32
0xE1-0xEF	reserved for ISO use as LargeExtensionDescriptor	32
0xF0-0xFF	user private as LargeExtensionDescriptor	32

8.4 Usage of the Object Descriptor Framework

8.4.1 Linking Scene Description and Object Descriptors

8.4.1.1 Associating Object Descriptors to Media Objects

A media object is associated to its elementary stream resources via an object descriptor. The object is used by the scene description by means of the `objectDescriptorID`, as specified in Subclause 9.3.1.8.9. Each media object has a specific type (audio, visual, inlined scene description, etc.). It shall only be associated to an object descriptor that advertises elementary streams that are compatible to the type of the media object.

The behavior of the terminal is undefined if an object descriptor advertises elementary streams with stream types that are incompatible with the associated media object.

8.4.1.2 Hierarchical scene and object description

The scene description allows a hierarchical description of a scene, using multiple scene description streams related to each other through inline nodes. The streaming resources for the content are in that case similarly described in hierarchically nested object descriptor streams. The association of the appropriate streams is done through object descriptors, as detailed further in Subclause 8.4.2.

Kommentar: This doesn't make sense. OD streams cannot be "hierarchically nested."

8.4.1.3 Name Scope Definition for Scene Description and Object Descriptor Streams

Object descriptors for scene description and object descriptor streams always group corresponding streams of both types in a single object descriptor, as specified in Subclause 8.4.2. Additionally, the `objectDescriptorID` and `ES_ID` identifiers that label the object descriptors and elementary streams, respectively, have a scope defined by the following rules:

- ES with `streamType` = ObjectDescriptorStream or SceneDescriptionStream that are associated to a single object descriptor form a common name scope for the `objectDescriptorID` and `ES_ID` values that are used within these streams.
- ES with `streamType` = ObjectDescriptorStream or SceneDescriptionStream that are not associated to the same object descriptor do not belong to the same name scope.

Kommentar: Comment from Carsten: ***(To Julien/Liam: Should that also scope the nodeIDs?)***

An initial object descriptor is a special case of this type of object descriptor and shall only contain streams of type ObjectDescriptorStream and SceneDescriptionStream following the restrictions defined in Subclause 8.4.2.

8.4.2 Associating Multiple Elementary Streams in a Single Object Descriptor

8.4.2.1 Object Descriptor as Grouping Mechanism

An object descriptor may contain descriptions of a list of elementary streams, i. e., multiple ES_Descriptors that relate to the same media object. This establishes a grouping mechanism that is further qualified by the ES_Descriptor syntax elements `streamDependenceFlag, dependsOn_ES_ID`, as well as `streamType`. The semantic rules for the association of elementary streams within one `ObjectDescriptor` (OD) are detailed below.

8.4.2.2 Associating Elementary Streams with Same Type

An OD shall only associate ESs with compatible `streamType`, i. e., ESs of visualStream, audioStream or SceneDescriptionStream type shall not be mixed within one OD. An OD shall never contain only ESs of `streamType` = ObjectDescriptorStream.

Multiple ESs within one OD with the same `streamType` of either audioStream, visualStream or SceneDescriptionStream which do not depend on other ESs shall convey alternative representations of the same content.

8.4.2.3 Associating Elementary Streams with Different Types

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

- An OD may contain zero or one additional ES of `streamType` = ObjectContentInfoStream. This ObjectContentInfoStream shall be valid for the content conveyed through the other ESs associated to this OD.
- An OD may contain zero or one additional ESs of `streamType` = ClockReferenceStream. (see Subclause 10.2.5) A ClockReferenceStream shall be valid for those ES within the name scope that refer to the ES_ID of this ClockReferenceStream in their SLConfigDescriptor.
- An OD that contains ESs of `streamType` = SceneDescriptionStream may contain any number of additional ESs with `streamType` = ObjectDescriptorStream.

8.4.2.4 Dependent Elementary Streams

An ES may depend on another ES associated to the same OD, indicated by a `dependsOn_ES_ID`. The semantic meaning of dependencies is determined by the type of the associated streams and is opaque at this layer.

Stream dependancies are governed by the following rules:

- For dependent ES of `streamType` equal to either audioStream, visualStream or SceneDescriptionStream the dependent ES shall have the same `streamType` as the independent ES. This implies that the dependent stream contains enhancement information to the one it depends on.
- An ES that flows upstream, as indicated by `DecoderConfigDescriptor.upStream` = 1 shall always depend upon another ES of the same `streamType` that has the `upStream` flag set to zero. This implies that this upstream is associated to the downstream it depends on.
- An ES with `streamType` = SceneDescriptionStream may depend on a stream with `streamType` = ObjectDescriptorStream. This implies that the ObjectDescriptorStream contains the object descriptors that are refered to by this SceneDescriptionStream.
  If this OD contains a second ES with `streamType` = SceneDescriptionStream that depends on the first SceneDescriptionStream this further implies that the object descriptors in the ObjectDescriptorStream are valid for this additional SceneDescriptionStream as well.
- The availability of the dependent stream is undefined if an ES_Descriptor for the stream it depends upon is not available.

8.4.3 Accessing ISO/IEC 14496 Content

8.4.3.1 Introduction

Content complying to this Final Committee Draft of International Standard is accessed through initial object descriptors that may be known through URLs or by means outside the scope of this specification. A selection of suitable subsets of the set of elementary streams that are part of such a content item may then be done. This process is further detailed below, followed by a number of walk throughs that specify the conceptual steps that need to be taken for content access.

Note: The DMIF Application Interface (DAI) specified in ISO/IEC FCD 14496-6 incorporates the functionality that is needed to implement the described content access procedures.

8.4.3.2 The Initial Object Descriptor

Initial object descriptors serve to access content represented using this Final Committee Draft of International Standard unconditionally. They convey information about the combination profiles required by the terminal to be able to process the described content. Initial object descriptors may be conveyed by means not defined in this specification or in an object descriptor stream, if they refer to hierarchically inlined content.

Kommentar: The hierachical structure only applies to BIFS inlining and nowhere else (e.g., ODs are not structured hierarchically).

Ordinary object descriptors that convey scene description and object descriptor streams do not carry profile information, and hence can only be used to access content if that information required or is obtained by other means. For example, if this object descriptor refers to a media object that is used in an inlined scene, then the profile indication is provided at the initial object descriptor used at the highest level of the inline hierarchy.

8.4.3.3 Selection of Elementary Streams for a Media Object

The selection of one or more ESs representing a single media object may be governed by the profile indications that are conveyed in the initial object descriptor and ES_Descriptors, respectively. In that case elementary streams with suitable object profiles that correspond to the initially signaled profile shall be available for selection. Additionally, streams that require more computing or bandwidth resources might be advertised in the object descriptor and may be used by the receiving terminal if it is capable of processing them.

In case streams do not indicate any profiles or if profile 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 media object 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.

To facilitate this approach, ESs should be ordered within an OD according to the content creator's preference. The ES that is first in the list of ES attached 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 receiver.

8.4.3.4 Usage of URLs in the Object Descriptor Framework

URLs in the object descriptor framework serve to locate either inlined content compliant with this Final Committee Draft of International Standard or the elementary stream data associated to individual media objects. URLs in ES_Descriptors imply that the complete description of the stream is available locally. URLs in object descriptors imply that the description of the resources for the associated media object or the inlined content is only available at the remote location. Note, however, that depending on the value of `includeInlineProfilesFlag` in the initial object descriptor, the global resources needed may already be known (i.e., including remote, inlined portions).

8.4.3.5 Accessing content through a known Object Descriptor

Kommentar: This starts to look too much like implementation issues. Perhaps statements "equivalent to the following" should be inserted.

8.4.3.5.1 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.

8.4.3.5.2 Content Access Procedure

1. The ES_ID for the streams that are to be opened are determined.
2. Requests for delivery of the selected ESs are made, using a suitable channel set up mechanism.
3. The channel set up mechanism shall return handles to the streams that correspond to the requested list of ES_IDs
4. In interactive scenarios, a confirmation that the terminal is ready to receive data is delivered to the sender.
5. Delivery of streams starts.
6. Scene description and object descriptor stream are read.
7. Further streams are opened as needed with the same procedure, starting at step 1.

8.4.3.6 Accessing content through a URL in an Object Desciptor

8.4.3.6.1 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.

8.4.3.6.2 Content access procedure

1. A connection to a 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 Subclause 8.4.3.5.2.

8.4.3.7 Accessing content through a URL in an Elementary Stream Descriptor

8.4.3.7.1 Pre-conditions

- An ES_Descriptor pointing to a stream through a URL has been aquired.
- A mechanism exists to open a channel that takes a URL as input and provides some returned data as output.

8.4.3.7.2 Content access procedure

1. The local ES_Descriptor specifies the configuration of the stream.
2. Request for delivery of the stream is made, using a channel set up call with the URL as parameter.
3. The channel set up call shall return a handle to the stream.
4. In interactive scenarios, a confirmation that the terminal is ready to receive data is delivered to the sender.
5. Delivery of stream starts.

8.4.3.8 Example of Content Access

The set up example in the followin 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 DMIF Application Interface (DAI), specified in Part 6 of this Final Committee Draft of International Standard, 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. A DA_ChannelReady() may optionally be sent to the remote side to start stream delivery.

Additional streams that are identified when processing the content of the object descriptor stream(s) are subsequently opened using the same procedure.

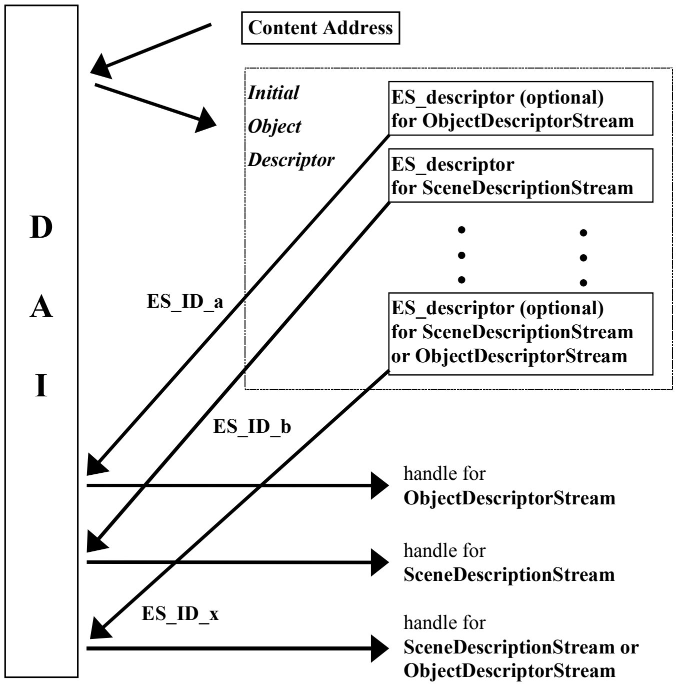


Figure 8-2: Content access example

8.4.3.9 Example of Complex Content Access

The example in Figure 8-3 shows a complex 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 in the object descriptor stream of the first scene. It 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.

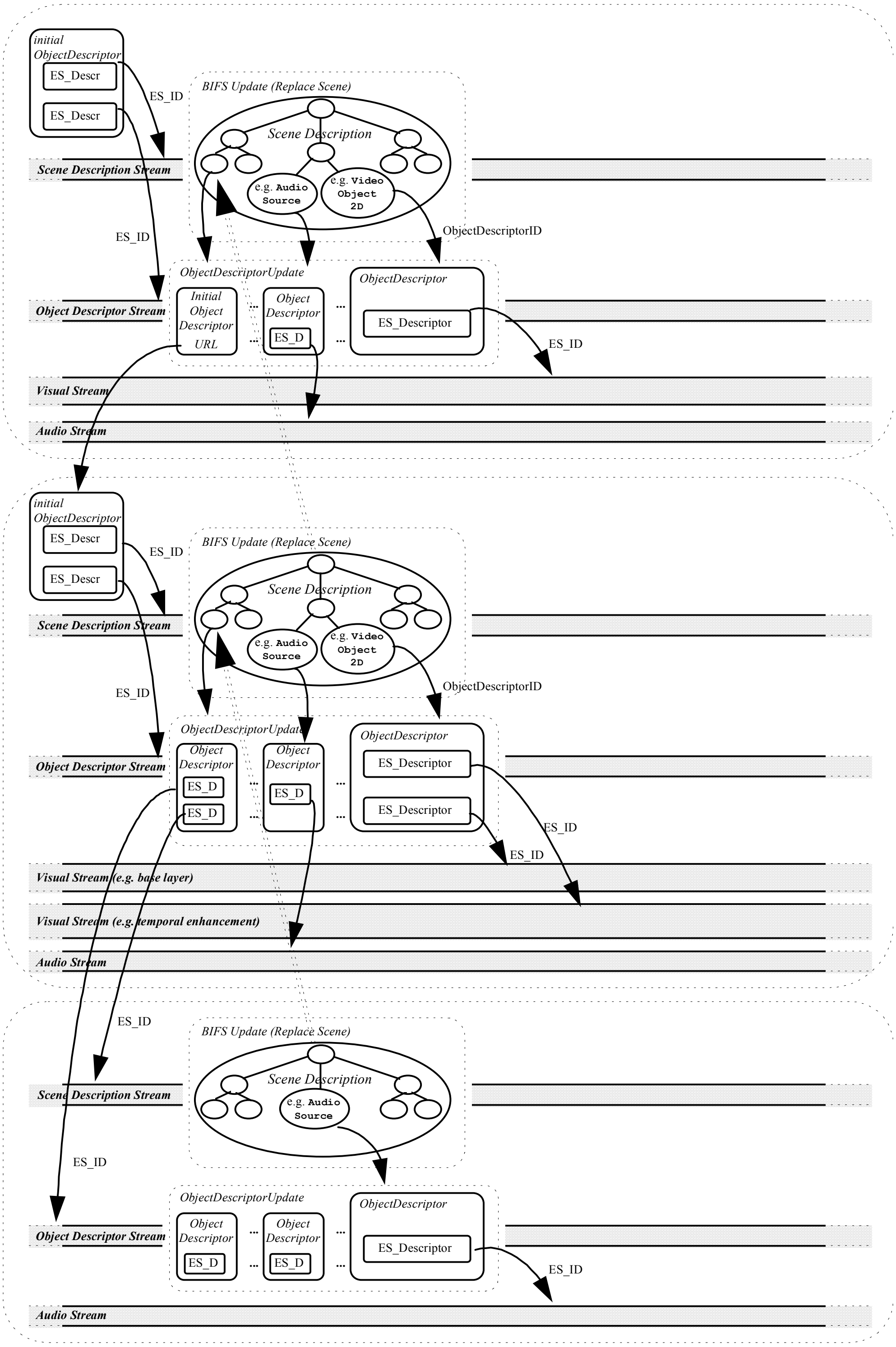


Figure 8-3: Complex content example

9. Scene Description

9.1 Introduction

9.1.1 Scope

This Final Committee Draft of International Standard addresses the coding of objects of various types: natural video and audio objects as well as textures, text, 2- and 3-dimensional graphics, and also synthetic music and sound effects. To reconstruct a multimedia scene at the terminal, it is hence no longer sufficient to just transmit the raw audiovisual data to a receiving terminal. Additional information is needed in order to combine these objects at the terminal and construct and present to the end user a meaningful multimedia scene. This information, called *scene description*, determines the placement of media objects in space and time and is transmitted together with the coded objects as illustrated in Figure 9-1. Note that the scene description only describes the structure of the scene. The action of putting these objects together in the same representation space is called *composition*. The action of transforming these media objects from a common representation space to a specific presentation device (i.e., speakers and a viewing window) is called *rendering*.

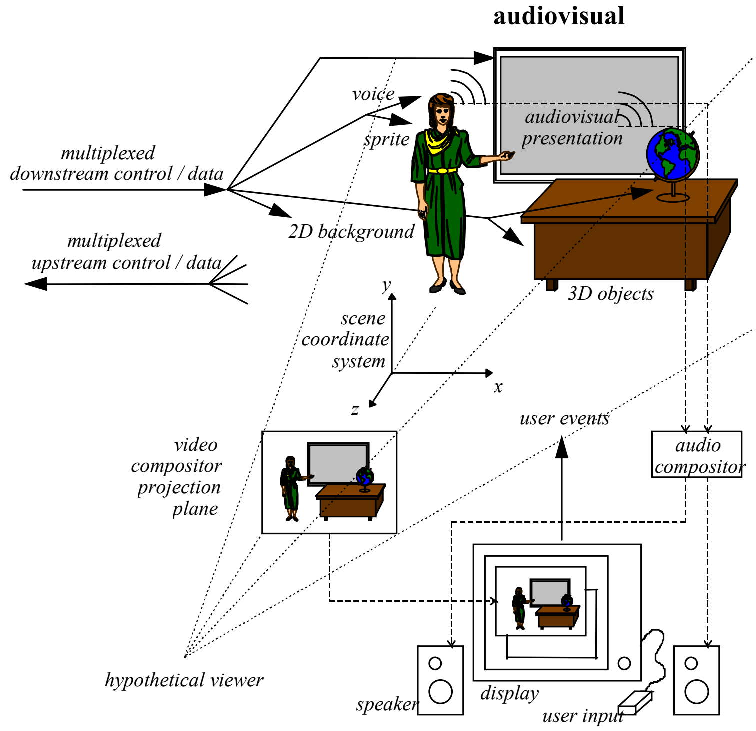


Figure 9-1: An example of an object-based multimedia scene

Independent coding of different objects may achieve higher compression, but it also brings the ability to manipulate content at the terminal. The behaviors of objects and their response to user inputs can thus also be represented in the scene description.

The scene description framework used in this Final Committee Draft of International Standard is based largely on ISO/IEC 14772-1 (Virtual Reality Modeling Language – VRML).

9.1.2 Composition

This specification defines only the syntax and semantics of bit streams that describe the spatio-temporal relationships of scene objects. It does not describe a particular way for a terminal to compose or render the scene, as these tasks are implementation-specific. The scene description representation is termed *"BInary Format for Scenes"* (BIFS).

9.1.3 Scene Description

In order to facilitate the development of authoring, editing and interaction tools, scene descriptions are coded independently from the audiovisual media that form part of the scene. This allows modification of the scene without having to decode or process in any way the audiovisual media.

The following subclauses detail the scene description capabilities that are provided by this Final Committee Draft of International Standard.

9.1.3.1 Grouping of objects

A scene description follows a hierarchical structure that can be represented as a tree. Nodes of the graph form *media objects*, as illustrated in Figure 9-2. The structure is not necessarily static; nodes may be added, deleted or be modified.

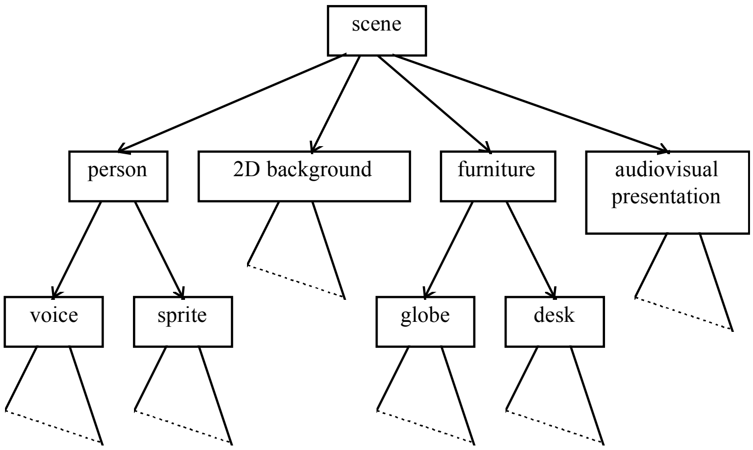


Figure 9-2: Logical structure of the scene

9.1.3.2 Spatio-Temporal positioning of objects

Media objects have both a spatial and a temporal extent. Complex objects are constructed by using appropriate scene description nodes that combine to form complex objects and thus build up the scene description tree. Objects may be located in 2-dimensional or 3-dimensional space. Each object has a local coordinate system. A local coordinate system is one in which the object has a fixed spatio-temporal location and scale (size and orientation). Objects are positioned in a scene by specifying a coordinate transformation from the object's local coordinate system into another coordinate system defined by a parent node in the tree.

9.1.3.3 Attribute value selection

Individual scene description nodes expose a set of parameters through which several aspects of their behavior can be controlled. Examples include the pitch of a sound, the color of a synthetic visual object, or the speed at which a video sequence is to be played.

A clear distinction should be made between the media object itself, the attributes that enable the control of its position and behavior, and any elementary streams that contain coded information representing some attributes of the object. A media object that has an associated elementary stream is called a *streamed media object*.

Example: A video object may be associated with an elementary stream that contains its coded representation, and also have a start time and end time as attributes attached to it.

This Final Committee Draft of International Standard also provides tools for enabling dynamic scene behavior and user interaction with the presented content. User interaction can be separated into two major categories: client-side and server-side. Only client side interactivity is addressed in this subclause, as it is an integral part of the scene description.

Client-side interaction involves content manipulation that is handled locally at the end-user's terminal. It consists of the modification of attributes of scene objects according to specified user actions. For example, a user can click on a scene to start an animation or video sequence. The facilities for describing such interactive behavior are part of the scene description, thus ensuring the same behavior in all terminal conforming to this Final Committee Draft of International Standard.

9.2 Concepts

9.2.1 Global Structure of BIFS

BIFS is a compact binary format representing a pre-defined set of scene objects and behaviors along with their spatio-temporal relationships. In particular, BIFS contains the following four types of information:

a) The attributes of media objects, which define their audio-visual properties.

b) The structure of the scene graph which contains these media objects.

c) The pre-defined spatio-temporal changes (or "self-behaviors") of these objects, independent of user input.

Example A sphere that rotates forever at a speed of 5 radians per second around a particular axis.

d) The spatio-temporal changes triggered by user interaction.

Example The activation of an animation when a user clicks on a given object.

These properties are intrinsic to the BIFS format. Further properties relate to the fact that BIFS data is itself conveyed to the receiver as an elementary stream. Portions of BIFS data that become valid at a given point in time are contained in BIFS CommandFrames (see Subclause 9.4.2.1) and are delivered within time-stamped access units as defined in Subclause 9.2.12.2. This allows modification of the scene description at given points in time by means of BIFS-Command or BIFS-Anim structures as specified in Subclause 9.2.17. It should be noted that even the initial BIFS scene is sent as a BIFS-Command. The semantics of a BIFS stream are specified in Subclause 9.2.12.

9.2.2 BIFS Scene Graph

Conceptually, BIFS scenes represent (as in ISO/IEC 14772-1) a set of visual and aural primitives distributed in a directed acyclic graph, in a 3D space. However, BIFS scenes may fall into several sub-categories representing particular cases of this conceptual model. In particular, BIFS scene descriptions support scenes composed of aural primitives as well as:

- 2D only primitives
- 3D only primitives
- A mix of 2D and 3D primitives, in several ways:
  1) Complete 2D and 3D scenes layered in a 2D space with depth
  2) 2D and 3D scenes used as texture maps for 2D or 3D primitives
  3) 2D scenes drawn in the local X Y plane of the local coordinate system in a 3D scene

The following figure describes a typical BIFS scene structure.

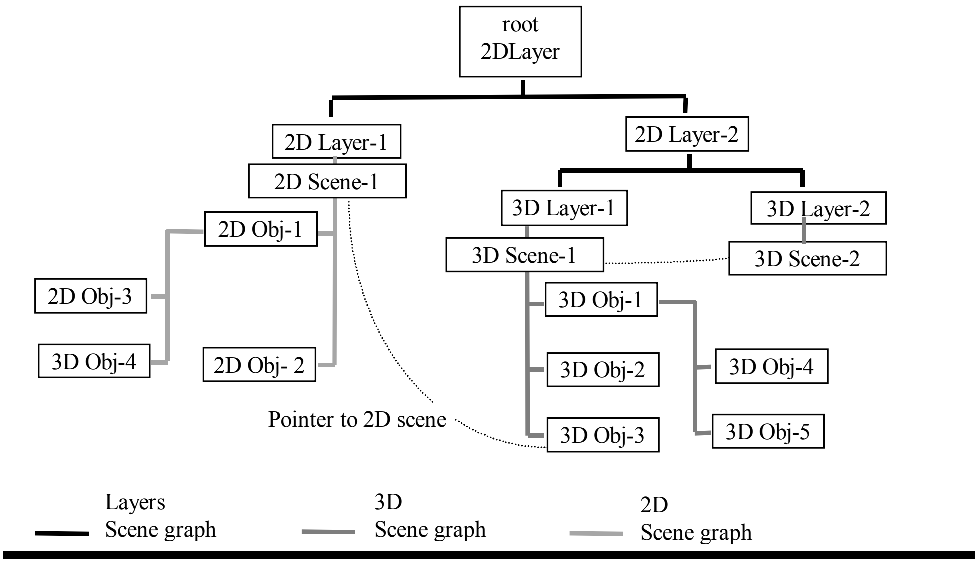


Figure 9-3: Scene graph example. The hierarchy of 3 different scene graphs is shown: a 2D graphics scene graph and two 3D graphics scene graphs combined with the 2D scene via layer nodes. As shown in the picture, the 3D Layer-2 is the same scene as 3D Layer-1, but the viewpoint may be different. The 3D Obj-3 is an Appearance node that uses the 2D Scene-1 as a texture node.

9.2.3 Standard Units

As described in ISO/IEC 14772-1, Subclause 4.4.5, the standard units used in the scene description are the following:

Category	Unit
Distance in 3D	Meter
Color Space	RGB [0,1] [0,1] [0,1]
Time	Seconds
Angle	Radians

Figure 9-4: Standard Units

9.2.4 2D Coordinate System

The origin of the 2D coordinate system is positioned in the center of the rendering area, the x-axis is positive to the right, and the y-axis is positive upwards.

The width of the rendering area represents -1.0 to +1.0 (meters) on the x-axis. The extent of the y-axis in the positive and negative directions is determined by the aspect ratio so that the unit of distance is equal in both directions. The rendering area is either the whole screen, when viewing a single 2D scene, or the rectangular area defined by the parent grouping node, or a **Composite2DTexture**, **CompositeMap** or **Layer2D** that contains a subordinate 2D scene description.

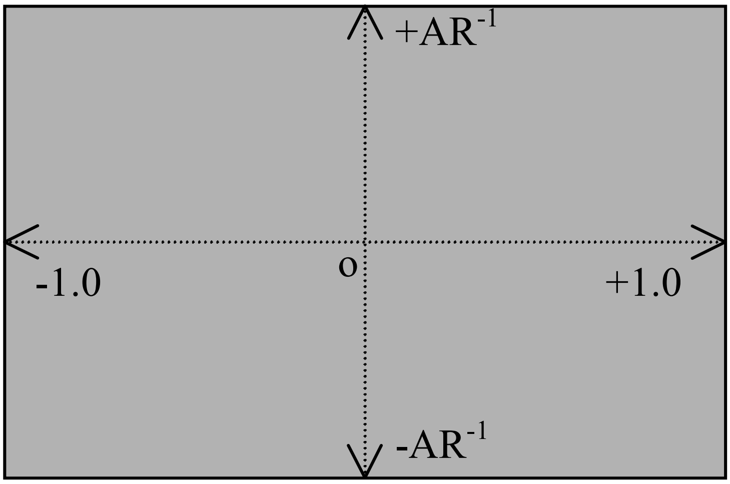


Figure 9-5: 2D Coordinate System (AR = Aspect Ratio)

In addition to meter-based metrics, it is also possible to use pixel-based metrics. In this case, 1 meter is set to be equal to the distance between two pixels. This applies to both the horizontal (x-axis) and vertical (y-axis) directions.

The selection of the appropriate metrics is performed by the content creator. In particular, it is controlled by the BIFSScene syntax (see Subclause 9.3.1.1): when the “isPixel” field is set to ‘1’ then pixel metrics are used for the entire scene..

Kommentar: I removed the reference to the BIFSScene syntax because, as fas as I know, it's now in the decoder config.

9.2.5 3D Coordinate System

The 3D coordinate system is as described in ISO/IEC 14772-1, Subclause 4.4.5.

9.2.6 Mapping of Scenes to Screens

BIFS scenes enable the use of still images and videos by copying, pixel by pixel, the output of the decoders to the screen. In this case, the same scene will appear different on screens with different resolutions. The nodes which facilitate this are **Image2D** (see 9.5.2.2.8) and **VideoObject2D** (see 9.5.2.2.23). BIFS scenes that do not use these primitives are independent from the screen on which they are viewed.

9.2.6.1 Transparency

Content complying with this Final Committee Draft of International Standard may include still images or video sequences with representations that include alpha values. These values provide transparency information and are to be treated as specified in ISO/IEC 14772-1, Subclause 4.14. The only exception is the use of video sequences represented according to Part 2 of this Final Committee Draft of International Standard is used. In this case, transparency is handled as specified in Part 2 of this Final Committee Draft of International Standard.

9.2.7 Nodes and fields

9.2.7.1 Nodes

The BIFS scene description consists of a collection of *nodes* that describe the scene structure. A media object in the scene is described by one or more nodes, which may be grouped together (using a grouping node). Nodes are grouped into Node Data Types and the exact type of the node is specified using a **nodeType** field.

A media object may be completely described within the BIFS information, e.g. **Box** with **Appearance**, or may also require elementary stream data from one or more audiovisual objects, e.g. **MovieTexture** or **AudioSource**. In the latter case, the node includes a reference to an object descriptor that indicates which elementary stream(s) is (are) associated with the node, or directly to a URL description (see ISO/IEC 14772-1, Subclause 4.5.2).

Kommentar: OK, a few sections above we said that each node is a scene object. Now we say that each object corresponds to one or more nodes. Later on, with the object descriptors, we say that an av object is a collection of elementary streams. Needs a pass to streamline these definitions.

9.2.7.2 Fields and Events

See ISO/IEC 14772-1, Subclause 5.1.

9.2.8 Basic Data Types

There are two general classes of fields and events; fields/events that contain a single value (e.g. a single number or a vector), and fields/events that contain multiple values. Multiple-valued fields/events have names that begin with MF, whereas single valued begin with SF.

9.2.8.1 Numerical data and string data types

9.2.8.1.1 Introduction

For each basic data type, single fields and multiple fields data types are defined in ISO/IEC 14772-1, Subclause 5.2. Some further restrictions are described herein.

9.2.8.1.2 SFInt32/MFInt32

When **ROUTE**ing values between two SF**Int32**s note shall be taken of the valid range of the destination. If the value being conveyed is outside the valid range, it shall be clipped to be equal to either the maximum or minimum value of the valid range, as follows:

if x > max, x := max

if x < min, x := min

9.2.8.1.3 SFTime

The **SFTime** field and event specifies a single time value. Time values shall consist of 64-bit floating point numbers indicating a duration in seconds or the number of seconds elapsed since the origin of time as defined in the semantics for each **SFTime** field.

9.2.8.2 Node data types

Nodes in the scene are also represented by a data type, namely SFNode and MFNode types. This Final Committee Draft of International Standard also defines a set of sub-types, such as SFColorNode, SFMaterialNode. These Node Data Types (NDTs) allow efficient binary representation of BIFS scenes, taking into account the usage context to achieve better compression. However, the generic SFNode and MFNode types are sufficient for internal representations of BIFS scenes.

9.2.9 Attaching nodeIDs to nodes

Each node in a BIFS scene graph may have a nodeID associated with it, to be used for referencing. ISO/IEC 14772-1, Subclause 4.6.2, describes the DEF statement which is used to attach names to nodes. In BIFS scenes, an integer value is used for the same purpose for nodeIDs. The number of bits used to represent these integer values is specified in the BIFS DecoderConfigDescriptor.

Kommentar: Where is this defined???

9.2.10 Using Pre-Defined Nodes

In the scene graph, nodes may be accessed for future changes of their fields. There are two main sources for changes of the values of BIFS nodes' fields:

a) The modifications occurring from the ROUTE mechanism, which enables the description of dynamic behaviors in the scene.
b) The modifications occurring from the BIFS update mechanism (see 9.2.17).

The mechanism for naming and reusing nodes is further described in ISO/IEC 14772-1, Subclause 4.6.3. Note that this mechanism results in a scene description that is a directed acyclic graph, rather than a simple tree.

The following restrictions apply:

a) Nodes are identified by the use of nodeIDs, which are binary numbers conveyed in the BIFS bitstream.
b) The scope of nodeIDs is given in Subclause 9.2.12.5.
c) No two nodes delivered in the same elementary stream may have the same nodeID.

9.2.11 Internal, ASCII and Binary Representation of Scenes

This Final Committee Draft of International Standard describes the attributes of media objects using node structures and fields. These fields can be one of several types (see 9.2.7.2). To facilitate animation of the content and modification of the objects' attributes in time, within the terminal, it is necessary to use an internal representation of nodes and fields as described in the node specifications (Subclause 9.5). This is essential to ensure deterministic behaviour in the terminal's compositor, for instance when applying ROUTEs or differentially coded BIFS-Anim frames. The observable behaviour of compliant decoders shall not be affected by the way in which they internally represent and transform data; i.e., they shall behave as if their internal representation is as defined herein.

However, at transmission time, different attributes need to be quantized or compressed appropriately. Thus, the binary representation of fields may differ according to the precision needed to represent a given Media Object, or according to the types of fields. The semantics of nodes are described in Subclause 9.5. The binary syntax which represents the binary format as transported in streams conforming to this Final Committee Draft of International Standard is provided in the Subclause 9.3 and uses the Node Coding Parameters provided in Annex H.

9.2.11.1 Binary Syntax Overview

9.2.11.1.1 Scene Description

The entire scene is represented by a binary encoding of the scene graph. This encoding restricts the VRML grammar as defined in ISO/IEC 14772-1, Annex A, but still enables the representation of any scene that can be generated by this grammar.

Example: One example of the grammatical differences is the fact that all ROUTEs are represented at the end of a BIFS scene, and that a global grouping node is required at the top level of the scene.

9.2.11.1.2 Node Description

Node types are encoded according to the context of the node.

9.2.11.1.3 Fields description

Fields are quantized whenever possible to improve compression efficiency. Several aspects of the dequantization process can be controlled by adjusting the parameters of the QuantizationParameter node.

9.2.11.1.4 ROUTE description

All ROUTEs are described at the end of the scene.

9.2.12 BIFS Elementary Streams

The BIFS scene description may, in general, be time variant. Consequently, BIFS data is carried in its own elementary stream and is subject to the provisions of the Systems Decoder Model (Clause 7).

9.2.12.1 BIFS-Command

BIFS data is encapsulated in BIFS-Command structures. For the detailed specification of all BIFS-Command structures see Subclause 9.3.2. Note that this does not imply that a BIFS-Command must contain a complete scene description.

9.2.12.2 BIFS Access Units

BIFS data is further composed of BIFS access units. An access unit contains one UpdateFrame that shall become valid (in an ideal compositor) at a specific point in time. This point in time is the composition time, which can be specified either explicitly or implicitly (see Sync Layer, Subclause 10.2.3).

9.2.12.3 Time base for the scene description

As with any elementary stream, the BIFS stream has an associated time base. The syntax to convey time bases to the receiver is specified in 10.2.4. It is also possible to indicate that the BIFS stream uses the time base of another elementary stream (see Subclause 10.2.3). All time stamps in the BIFS are expressed in SFTime format but refer to this time base.

9.2.12.4 Composition Time Stamp semantics for BIFS Access Units

The CTS of a BIFS access unit indicates the point in time that the BIFS description in this access unit becomes valid (in an ideal compositor). This means that any audiovisual objects that are described in the BIFS access unit will ideally become visible or audible exactly at this time unless a different behavior is specified by the fields of their nodes.

9.2.12.5 Multiple BIFS streams

Scene description data may be conveyed in more than one BIFS elementary streams. This is indicated by the presence of one or more Inline/Inline2D nodes in a BIFS scene description that refer to further elementary streams as specified in Subclauses 9.5.3.3.15 and 9.5.2.2.11. Therefore, multiple BIFS streams have a hierarchical dependency. Note, however, that it is not required that all BIFS streams adhere to the same time base.

Example: An application of hierarchical BIFS streams is a multi-user virtual conferencing scene, where subscenes originate from different sources.

The scope for names (nodeID, ROUTEID, objectDescriptorID) used in a BIFS stream is given by the grouping of BIFS streams within one Object Descriptor (see Subclause 8.4.1.3). Conversely, BIFS streams that are not declared in the same Object Descriptor form separate name spaces. As a consequence, an Inline node always opens a new name space that is populated with data from one or more BIFS streams. It is not possible to reference parts of the scene outside the name scope of the BIFS stream.

9.2.12.6 Time fields in BIFS nodes

In addition to the composition time stamps that specify the activation of BIFS access units, several BIFS nodes have fields of type SFTime that identify a point in time at which an event occurs (change of a parameter value, start of a media stream, etc). These fields are time stamps relative to the time base that applies to the BIFS elementary stream that has conveyed the respective nodes. More specifically, this means that any time instant, and therefore time duration, is unambiguously specified.

SFTime fields of some nodes require absolute time values. Absolute time (“wall clock” time) cannot be directly derived through knowledge of the time base, since time base ticks need not have a defined relation to the wall clock. However, the absolute time can be related to the time base if the wall clock time that corresponds to the composition time stamp of the BIFS access unit that has conveyed the respective BIFS node is known. This is achieved by an optional wallClockTimeStamp as specified in Subclause 10.2.4. Following receipt of one such time association, all absolute time references within this BIFS stream can be resolved.

Note: *The SFTime fields that define the start or stop of a media stream are relative to the BIFS time base. If the time base of the media stream is a different one, it is not generally possible to set a startTime that corresponds exactly to the Composition Time of a Composition Unit of this media stream.*

Example: The example below shows a BIFS access unit that is to become valid at CTS. It conveys a media node that has an associated elementary stream. Additionally, it includes a **MediaTimeSensor** that indicates an **elapsedTime** that is relative to the CTS of the BIFS access unit. Finally, a ROUTE definition routes Time=(now) to the **startTime** of the media node when the **elapsedTime** of the MediaTimeSensor has passed. The Composition Unit (CU) that is available at that time CTS+MediaTimeSensor.elapsedTime is the first one available for composition.

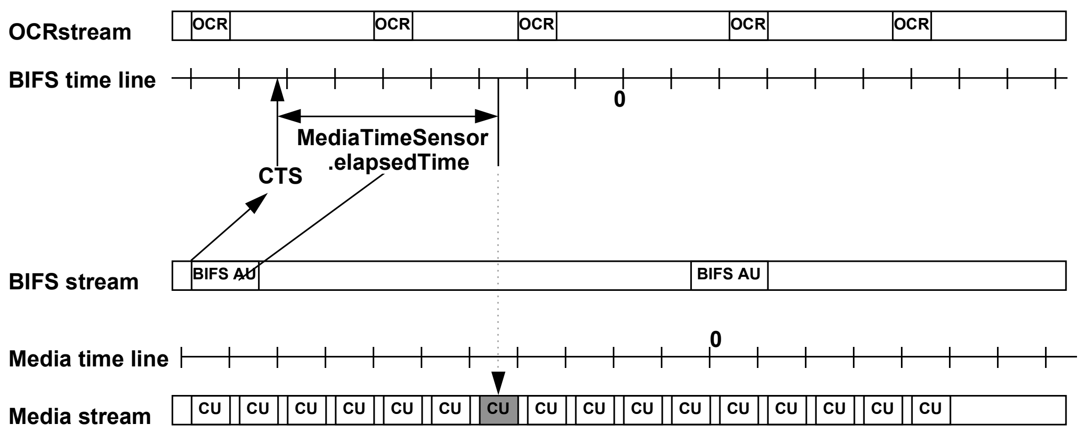


Figure 9-6: Media start times and CTS

9.2.12.7 Time events based on media time

Regular **SFTime** time values in the scene description can be used for triggering events based on the BIFS time base. In order to be able to trigger events in the scene at a specific point on the media time line, a **MediaTimeSensor** node is specified in 9.5.1.2.7.

9.2.13 Time-dependent nodes

The semantics of the **exposedFields**: **startTime**, **stopTime**, and **loop**, and the **eventOut**: **isActive** in time-dependent nodes are as described in ISO/IEC 14772-1, Subclause 4.6.9. This Final Committee Draft of International Standard adds the following time-dependent nodes: **AnimationStream**, **AudioSource**, **MediaTimeSensor, VideoObject2D**. The semantics of time and **SFTime** fields in BIFS are described in 9.2.12.6.

9.2.14 Audio

9.2.14.1 Audio sub-trees

Audio nodes are used for building audio scenes in the decoder terminal from audio sources coded with tools specified in this Final Committee Draft of International Standard. The audio scene description capabilities provide two functionalities:

- "Physical modelling" composition for virtual-reality applications, where the goal is to recreate the acoustic space of a real or virtual environment.
- "Post-production" composition for traditional content applications, where the goal is to apply high-quality signal processing transformations.

Audio may be included in either 2D or 3D scene graphs. In a 3D scene, the audio may be spatially presented to appear as originating from a particular 3D direction, according to the positions of the object and the listener.

The **Sound** node is used to attach audio to 3D scene graphs and the **Sound2D** node is used to attach audio to 2D scene graphs. As with visual objects, an audio object represented by one of these nodes has a position in space and time, and is transformed by the spatial and grouping transforms of nodes hierarchically above it in the scene.

The nodes *below* the **Sound/Sound2D** nodes, however, constitute an *audio sub-tree.* This subtree is used to describe a particular audio object through the mixing and processing of several audio streams. Rather than representing a hierarchy of spatio-temporal transformations, the nodes within the audio subtree represent a signal flow graph that describes how to create the audio object from the audio coded in the **AudioSource** streams. That is, each audio subtree node (**AudioSource**, **AudioMix**, **AudioSwitch**, **AudioFX, AudioClip, AudioDelay)** accepts one or several channels of input audio, and describes how to turn these channels of input audio into one or more channels of output. The only sounds presented in the audiovisual scene are those which are the output of audio nodes that are children of a **Sound/Sound2D** node (that is, the "highest" outputs in the audio sub-tree). The remaining nodes represent "intermediate results" in the sound computation process and the sound represented therein is presented to the user.

The normative semantics of each of the audio subtree nodes describe the exact manner in which to compute the output audio the input audio for each node based on its parameters.

9.2.14.2 Overview of sound node semantics

This subclause describes the concepts for normative calculation of the audio objects in the scene in detail, and describes the normative procedure for calculating the audio signal which is the output of a **Sound** node given the audio signals which are its input.

Recall that the audio nodes present in an audio subtree do not each represent a sound to be presented in the scene. Rather, the audio sub-tree represents a signal-flow graph which computes a single (possibly multichannel) audio object based on a set of audio inputs (in **AudioSource** nodes) and parametric transformations. The only sounds which are presented to the listener are those which are the "output" of these audio sub-trees, as connected to a **Sound/Sound2D** node. This subclause describes the proper computation of this signal-flow graph and resulting audio object.

As each audio source is decoded, it produces data that is stored in Composition Memory (CM). At a particular time instant in the scene, the compositor shall receive from each audio decoder a CM such that the decoded time of the first audio sample of the CM for each audio source is the same (that is, the first sample is synchronized at this time instant). Each CM will have a certain length, depending on the sampling rate of the audio source and the clock rate of the system. In addition, each CB has a certain number of channels, depending on the audio source.

Kommentar: Eric Scheirer says that some further comments on this will come from the ASI AHG work.

Each node in the audio sub-tree has an associated *input buffer* and *output buffer*, except for the **AudioSource** node which has no input buffer. The CM for the audio source acts as the input buffer of audio for the **AudioSource** with which the decoder is associated. As with CM, each input and output buffer for each node has a certain length, and a certain number of channels.

As the signal-flow graph computation proceeds, the output buffer of each node is placed in the *input buffer* of its parent node, as follows:

If a audio node **N** has n children, and each of the children produces $k(i)$ channels of output, for $1 <= i <= n$, then the node **N** shall have $k(1) + k(2) + ... + k(n)$ channels of input, where the first $k(1)$ channels [number 1 through $k(1)$] shall be the channels of the first child, the next $k(2)$ channels [number $k(1)+1$ through $k(1)+k(2)$] shall be the channels of the second child, and so forth.

Then, the output buffer of the node is calculated from the input buffer based on the particular rules for that node.

9.2.14.2.1 Sample-rate conversion

If the various children of a **Sound** node do not produce output at the same sampling rate, then the lengths of the output buffers of the children do not match, and the sampling rates of the childrens' output must be brought into alignment in order to place their output buffers in the input buffer of the parent node. The sampling rate of the input buffer for the node shall be the fastest of the sampling rates of the children. The output buffers of the children shall be resampled to be at this sampling rate. The particular method of resampling is non-normative, but the quality shall be at least as high as that of quadratic interpolation, that is, the noise power level due to the interpolation shall be no more than –12dB relative to the power of the signal. Implementors are encouraged to build the most sophisticated resampling capability possible into terminals.

The output sampling rate of a node shall be the output sampling rate of the input buffers after this resampling procedure is applied.

Content authors are advised that content which contains audio sources operating at many different sampling rates, especially sampling rates which are not related by simple rational values, may produce scenes with a high computational complexity.

Example: Suppose that node N has children M1 and M2, all three audio nodes, and that M1 and M2 produce output at S1 and S2 sampling rates respectively, where S1 > S2. Then if the decoding frame rate is F frames per second, then

M1's output buffer will contain S1/F samples of data, and M2's output buffer will contain S2/F samples of data. Then, since M1 is the faster of the children, its output buffer values are placed in the input buffer of N. The output buffer of M2 is resampled by the factor S1/S2 to be S1/F samples long, and these values are placed in the input buffer of N. The output sampling rate of N is S1.

9.2.14.2.2 Number of output channels

If the **numChan** field of an audio node, which indicates the number of output channels, differs from the number of channels produced according to the calculation procedure in the node description, or if the **numChan** field of an **AudioSource** node differs in value from the number of channels of an input audio stream, then the **numChan** field shall take precedence when including the source in the audio sub-tree calculation, as follows:

a) If the value of the **numChan** field is strictly less than the number of channels produced, then only the first **numChan** channels shall be used in the output buffer.

b) If the value of the **numChan** field is strictly greater than the number of channels produced, then the "extra" channels shall be set to all 0's in the output buffer.

9.2.14.3 Audio-specific BIFS Nodes

In the following table, nodes that are related to audio scene description are listed.

Table 9-1: Audio-Specific BIFS Nodes

Node	Purpose	Subclause
AudioClip	Insert an audio clip to scene	9.5.1.3.2
AudioDelay	Insert delay to sound	9.5.1.2.2
AudioMix	Mix sounds	9.5.1.2.3
AudioSource	Define audio source input to scene	9.5.1.2.4
AudioFX	Attach structured audio objects to sound	9.5.1.2.5
AudioSwitch	Switching of audio sources in scene	9.5.1.2.6
Group, Group2D	Grouping of nodes and subtrees in a scene	9.5.2.2.7, 9.5.3.3.12
ListeningPoint	Define listening point in a scene	9.5.3.2.1
Sound, Sound2D	Define properties of sound	9.5.3.3.25, 9.5.2.2.20

9.2.15 Drawing Order

2D scenes are considered to have zero depth. Nonetheless, it is important to be able to specify the order in which 2D objects are composed. For this purpose, the **Transform2D** node contains a **field** called **drawingOrder**.

The **drawingOrder** field provides a mechanism for explicitly specifying the order in which 2D objects are drawn. **drawingOrder** values are floating point numbers and may be negative. By default, **drawingOrder** for all 2D objects is 0. The following rules determine the drawing order, including conflict resolution for objects having the same **drawingOrder**:

1. The object having the lowest **drawingOrder** shall be drawn first (taking into account negative values).
2. Objects having the same **drawingOrder** shall be drawn in the order in which they appear in the scene description.

The scope of drawing orders, explicit and implicit, is limited to the sub-scene to which they may belong. Note that sub-scenes, as a whole, have a drawing order within the higher level scene or sub-scene to which they belong.

9.2.16 Bounding Boxes

Some nodes have bounding box fields. The bounding box gives rendering hints to the implementation and need not be specified. However, when a bounding box is specified, it shall be large enough to enclose the geometry to which it pertains but need not be the bounding box of minimum dimensions.

The bounding box dimensions are specified by two fields. The **bboxCenter** specifies the point, in the node's local coordinate system, about which the box is centered. The **bboxSize** fields specify the bounding box size. A default **bboxSize** value of (-1, -1) in 2D or (-1, -1, -1) in 3D, implies that the bounding box is not specified and, if required, should be calculated by the terminal.

9.2.17 Sources of modification to the scene

9.2.17.1 Interactivity and behaviors

To describe interactivity and behavior of scene objects, the event architecture defined in ISO/IEC 14772-1, Subclause 4.10, is used. Sensors and ROUTEs describe interactivity and behaviors. Sensor nodes generate events based on user interaction or a change in the scene. These events are ROUTEd to Interpolator or other nodes to change the attributes of these nodes. If ROUTEd to an Interpolator, a new parameter is interpolated according to the input value, and is finally ROUTEd to the node which must process the event

9.2.17.1.1 Attaching ROUTEIDs to ROUTEs

ROUTEIDs may be attached to ROUTEs using the DEF mechanism, described in ISO/IEC 14772-1, Subclause 4.6.2. This allows ROUTEs to be subsequently referenced in BIFS-Command structures. ROUTEIDs are integer values and the namespace for ROUTEs is distinct from that of nodeIDs. The number of bits used to represent these integer values is specified in the BIFS DecoderConfigDescriptor.

Kommentar: Why isn't this configurable via the DecoderConfigDescriptor? The nodeID is.

The scope of ROUTEIDs is defined in Subclause 9.2.12.5 and no two ROUTEIDs transmitted in a single elementary stream my have the same ROUTEID.

Kommentar: Insert correct reference.

Note that the USE mechanism may not be used with ROUTEs.

9.2.17.2 External modification of the scene: BIFS-Commands

The BIFS-Command mechanism enables the change of any property of the scene graph. For instance, **Transform** nodes can be modified to move objects in space, **Material** nodes can be changed to modify an object's appearance, and fields of geometric nodes can be totally or partially changed to modify the geometry of objects. Finally, nodes and behaviors can be added or removed.

9.2.17.2.1 Overview

BIFS commands are used to modify a set of properties of the scene at a given time instant in time. However, for continuous changes of the parameters of the scene, the animation scheme described in the following subclause can be used. Commands are grouped into Update Frames in order to be able to send several commands in a single access unit. The following four basic commands are defined:

1. Insertion
2. Deletion
3. Replacement
4. Replacement of an entire scene

The first three commands can be used to update the following structures:

1. A node
2. A field or an indexed value in a multiple field; or
3. A ROUTE.

In addition, the 'Replacement' command can be used to replace a field.

Insertion of an indexed value in a field implies that all later values in the field have their indices incremented and the length of the field increases accordingly. Appending a value to an indexed value field also increases the length of the field but the indices of existing values in the field do not change.

Deletion of an indexed value in a field implies that all later values in the field have their indices decremented and the length of the field decreases accordingly.

Deletion of an indexed value in a field implies that all later values in the field have their indices decremented and the length of the field decreases accordingly.

The replacement of the whole scene requires a node tree representing a valid BIFS scene. The SceneReplace command is the only random access point in the BIFS stream.

In order to modify the scene the sender must transmit a BIFS-Command frame that contains one or more update commands. The identification of a node in the scene is provided by a nodeID. It should be noted that it is the sender's responsibility to provide this nodeID, which must be unique (see 9.2.12.5). A single source of updates is assumed. The identification of node fields is provided by sending their fieldIndex, its position in the fields list of that node.

The time of application of the update is the the composition time stamp, as defined in the Sync Layer (see Subclause 10.2.4).

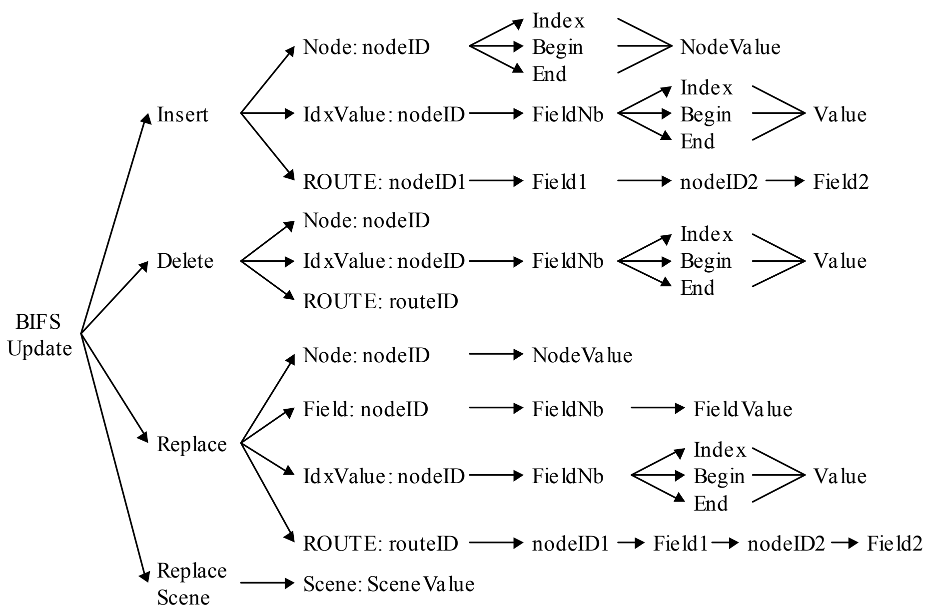


Figure 9-7: BIFS-Command Types

9.2.17.3 External animation of the scene: BIFS-Anim

BIFS-Command provides a mechanism for describing changes in the scene. An alternative facility called "BIFS-Anim" is also provided, that specifically addresses the continuous update of the fields of a particular node. BIFS-Anim is used to integrate different kinds of animation, including the ability to animate face models as well as meshes, 2D and 3D positions, rotations, scale factors, and color attributes. The BIFS-Anim information is conveyed in its own elementary stream.

9.2.17.3.1 Overview

BIFS-Anim elementary streams carry the following parameters (in the order shown):

1. Configuration parameters of the animation, also called the "Animation Mask".

   These describe the fields for which parameters are conveyed in the BIFS-Anim elementary stream, and also specify their quantization and compression parameters. Only eventIn or exposedField fields that have a dynID can be modified using BIFS-Anim. Such fields are called "dynamic" fields. In addition, the animated field must be part of an updateable node, i.e., a node that has been assigned a nodeID. The Animation Mask is composed of several elementary masks defining these parameters.

2. Animation parameters are sent as a set of "Animation Frames".

   An Animation Frame contains all the new values of the animated parameters at a specified time, except if it is specified that, for some frames, these parameters are not sent. The parameters can be sent in Intra (the absolute value is sent) and Predictive modes (the *difference* between the current and previous values is sent).

Animation parameters can be applied to any field of any updateable node of a scene which has an assigned dynID. The types of dynamic fields are all scalar types, single or multiple:

1. SFInt32/MFInt32
2. SFFloat/MFFloat
3. SFRotation/MFRotation
4. SFColor/MFColor

9.2.17.3.2 Animation Mask

Animation Masks represent the fields that are contained in the animation frames for a given animated node, and their associated quantization parameters. For each node of a scene that is to be animated, an Elementary Mask is transmitted that specifies these parameters.

9.2.17.3.3 Animation Frames

Animation Frames specify the values of the dynamic fields of updateable nodes that are being animated in BIFS-Anim streams.

9.3 BIFS Syntax

The binary syntax of all BIFS related information is described in this clause. For details on the semantic and decoding process of these structures, refer to the following subclause (9.4).

9.3.1 BIFS Scene and Nodes Syntax

The scene syntax defines the structure of a complete scene description.

9.3.1.1 BIFSConfig

```
class BIFSConfig {
   unsigned int(5) nodeIDbits;
   unsigned int(5) routeIDbits;
   bit(1) isCommandStream;
   if(isCommandStream) {
      bit(1) pixelMetric;
      bit(1) hasSize;
      if(hasSize) {
         uint(16) pixelWidth;
         unit(16) pixelHeight;
      }
   }
   else {
      AnimationMask animMask(nodeIDbits);
   }
}
```

9.3.1.2 BIFSScene

```
class BIFSScene(BIFSConfig cfg) {
   SFNode nodes(cfg, SFTopNode); // Description of the nodes
   bit(1) hasROUTEs;
   if (hasROUTEs) {
      ROUTEs routes(cfg);         // Description of the ROUTEs
```

```
   }
}
```

9.3.1.3 SFNode

```
class SFNode(BIFSConfig cfg, int nodeDataType) {
   bit(1) isReused ;    // This bit describes whether this node is
                        // a reused node or a newly defined one.
                        // This is equivalent to the VRML USE statement.
   if (isReused) {
      bit(cfg.nodeIDbits) nodeID;   // The node to be re-used
   }
   else {
      bit(ndt[nodeDataType].nbits) nodeType;
                        // This represents the type of the node.
      bit(1) isUpdateable;
                        // if updateable or can be reused, it needs ID
      if (isUpdateable) {
         bit(cfg.nodeIDbits) nodeID;
      }
      bit(1) MaskAccess;
      if (MaskAccess) {
         MaskNodeDescription mnode(GetNCT(nodeDataType, nodeType));
      }
      else {
         ListNodeDescription lnode(GetNCT(nodeDataType, nodeType));
      }
   }
}
```

9.3.1.4 MaskNodeDescription

```
class MaskNodeDescription(NCT nct) {
   for (i=0; i<nct.numDEFfields; i++) {
      bit(1) Mask;
      if (Mask)
         Field value(nct, i);
            // get field value using nodeType and field reference
            // to find out the appropriate field type.
            // This field can itself be a new node
   }
}
```

9.3.1.5 ListNodeDescription

```
class ListNodeDescription (NCT nct) {
   bit(1) endFlag;
   while (!EndFlag){
      int(nct.nDEFbits) fieldRef;
            // this identifies the field for this nodeType.
            // Length is derived from the number of fields
            // in the node of type defField and is the number
            // of bits necessary to represent the field number
      Field value(nct, fieldRef);
            // get field value using nodeType and fieldRef to
            // find out the appropriate field type. This field
            // can itself be a new node.
      bit(1) endFlag;
   }
}
```

9.3.1.6 Field

```
class Field(NCT nct, int fieldRef) {
   if (isSF(fieldType))    // is this an SF* type?
```

```
      SFField svalue(nct, fieldRef);
   else
      MFField mvalue(nct, fieldRef);
}
```

9.3.1.7 MFField

```
class MFField(NCT nct, int fieldRef)  {
   bit(1) isListDescription;
   if (isListDescription)
      MFListDescription lfield(nct, fieldRef);
   else
      MFVectorDescription vfield(nct, fieldRef);
}

class MFListDescription(NCT nct, int fieldRef)  {
   bit(1) endFlag;
   while (!endFlag) {
      SField field(nct, fieldRef);
      bit(1) endFlag;
   }
}

class MFVectorDescription(NCT nct, int fieldRef) {
   int(5) NbBits;       // number of bits for the number of fields
   int(NbBits) numberOfFields;
   SFField field[numberOfFields](nct, fieldRef);
}
```

9.3.1.8 SFField

```
class SFField(NCT nct, int fieldRef) {
   switch (nct.fieldType[fieldRef]) {

   case SFNodeType:
      SFNode nValue(nct.fieldType[fieldRef]);
      break;

   case SFBoolType:
      SFBool bValue;
      break;

   case SFColorType:
      SFColor cValue(nct, fieldRef);
      break;

   case SFFloatType:
      SFFloat fValue(nct, fieldRef);
      break;

   case SFInt32Type:
      SFInt32 iValue(nct, fieldRef);
      break;

   case SFRotationType:
      SFRotation rValue(nct, fieldRef);
      break;

   case SFStringType:
      SFString sValue;
      break;

   case SFTimeType:
      SFTime tValue;
      break;
```

```
   case SFUrlType:
      SFUrl uValue;
      break;

   case SFVec2fType:
      SFVec2f v2Value(nct, fieldRef);
      break;

   case SFVec3fType:
      SFVec3f v3Value(nct, fieldRef);
      break;
   }
}
```

9.3.1.8.1 Float

```
class Float {
   bit(1) useEfficientCoding;
   if (!useEfficientCoding)
      float(32) value;
   else {
      unsigned int(4) mantissaLength;
      if (mantissaLength == 0)
         float(0) value = 0;  // no mantissa implies value := 0
      else {
         int(3) exponentLength;
         int(mantissaLength) mantissa; // signed 2's complement
         if (exponentLength > 0)
            int(exponentLength) exponent; // signed 2's complement
         float(0) value = mantissa * 2 ^ exponent;
               // note: if exponent is not parsed, it is set to 0
      }
   }
}
```

9.3.1.8.2 SFBool

```
class SFBool {
   bit(1) value;
}
```

9.3.1.8.3 SFColor

```
class SFColor(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct, fieldRef);
   else {
      Float rValue;
      Float gValue;
      Float bValue;
   }
}
```

9.3.1.8.4 SFFloat

```
class SFFloat(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct, fieldRef);
   else
      Float value;
}
```

9.3.1.8.5 SFInt32

```
class SFInt32(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct, fieldRef);
   else
      int(32) value;
}
```

9.3.1.8.6 SFRotation

```
class SFRotation(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct, fieldRef);
   else {
      Float xAxis;
      Float yAxis;
      Float zAxis;
      Float angle;
   }
}
```

9.3.1.8.7 SFString

```
class SFString {
   unsigned int(5) lengthBits;
   unsigned int(lengthBits) length;
   char(8) value[length];
}
```

9.3.1.8.8 SFTime

```
class SFTime {
   double(64) value;
}
```

9.3.1.8.9 SFUrl

```
class SFUrl {
   bit(1) isOD;
   if (isOD)
      bit(10) ODid;
   else
      SFString urlValue;
}
```

9.3.1.8.10 SFVec2f

```
class SFVec2f(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct, fieldRef);
   else {
      Float value1;
      Float value2;
   }
}
```

9.3.1.8.11 SFVec3f

```
class SFVec3f(NCT nct, int fieldRef) {
```

```
   LQP lqp=getCurrentLQP();

   if (lqp!=0 && lqp.isQuantized[fieldRef])
      QuantizedField qvalue(nct,fieldRef);
   else {
      Float value1;
      Float value2;
      Float value3;
   }
}
```

9.3.1.9 QuantizedField

```
class QuantizedField(NCT nct, int fieldRef) {
   LQP lqp=getCurrentLQP();
   switch (nct.quantType[fieldRef]) {

   case 1:  // 3D Positions
      int(lqp.position3DNbBits) x;
      int(lqp.position3DNbBits) y;
      int(lqp.position3DNbBits) z;
      break;

   case 2:  // 2D Positions
      int(lqp.position2DNbBits) x;
      int(lqp.position2DNbBits) y;
      break;

   case 3:  // Drawing Order
      int(lqp.drawOrderNbBits) d;
      break;

   case 4:  // Color
      switch (nct.fieldType[fieldRef]) {
      case SFColor:
         int(lqp.colorNbBits) r;
         int(lqp.colorNbBits) g;
         int(lqp.colorNbBits) b;
         break;
      case SFFloat:
         int(lqp.colorNbBits) value;
         break;
      }
      break;

   case 5:  //TextureCoordinate
      int(lqp.textureCoordinateNbBits) xPos;
      int(lqp.textureCoordinateNbBits) yPos;
      break;

   case 6:  // Angle
      int(lqp.angleNbBits) angle;
      break;

   case 7:  // Scale
      switch (nct.fieldType[fieldRef])) {
      case SFVec2fType:
         int(lqp.scaleNbBits) xScale;
         int(lqp.scaleNbBits) yScale;
         break;
      case SFVec3fType:
         int(lqp.scaleNbBits) xScale;
         int(lqp.scaleNbBits) yScale;
         int(lqp.scaleNbBits) zScale;
         break;
      }
```

```
      break;

   case 8:  //Interpolator Key
      int(lqp.keyNbBits) key;
      break;

   case 9:  // Normal
      int(3) octantNb;
      int(2) triantNb;
      int(lqp.normalNbBits) xPos;
      int(lqp.normalNbBits) yPos;
      break;

   case 10: // Rotation
      int(3) octantNb;
      int(2) triantNb;
      int(lqp.normalNbBits) xPos;
      int(lqp.normalNbBits) yPos;
      int(lqp.angleNbBits) angle;
      break;

   case 11: // Object Size 3D
      int(lqp.position3DNbBits) size;
      break;

   case 12: // Object Size 2D
      int(lqp.position2DNbBits) size;
      break;

   case 13: // Linear Quantization
      switch (nct.fieldType[fieldRef])) {
      case SFVec2fType:
         int(nct.nbBits[fieldRef]) value1;
         int(nct.nbBits[fieldRef]) value2;
         break;
      case SFVec3fType:
         int(nct.nbBits[fieldRef]) value1;
         int(nct.nbBits[fieldRef]) value2;
         int(nct.nbBits[fieldRef]) value3;
         break;
      default:
         int(nct.nbBits[fieldRef]) value;
         break;
      }
      break ;
   }
}
```

9.3.1.10 ROUTE syntax

9.3.1.10.1 ROUTEs

```
class ROUTEs(BIFSConfig cfg) {
   bit(1) ListDescription;
   if (ListDescription)
      ListROUTEs lroutes(cfg);
   else
      VectorROUTEs vroutes(cfg);
}
```

9.3.1.10.2 ListROUTEs

```
class ListROUTEs(BIFSConfig cfg) {
   do {
      ROUTE route(cfg);
```

```
      bit(1) moreROUTEs;
   }
   while (moreROUTEs);
}
```

9.3.1.10.3 VectorROUTEs

```
class VectorROUTEs(BIFSConfig cfg) {
   int(5) nBits;
   int(nBits) length;
   ROUTE route[length](cfg);
}
```

9.3.1.10.3.1 ROUTE

```
class ROUTE(BIFSConfig cfg) {
   bit(1) isUpdateable;
   if (isUpdateable)
      bit(cfg.routeIDbits) routeID;
   bit(cfg.nodeIDbits) outNodeID;
   NCT nctOUT=GetNCTFromID(outNodeID);
   int(nctOUT.nOUTbits) outFieldRef;
      // this identifies the field for the node corresponding
      // to the outNodeID
   bit(cfg.nodeIDbits) inNodeID;
   NCT nctIN = GetNCTFromID(inNodeID);
   int(nctIN.nINbits) inFieldRef;
      // similar to outFieldRef
}
```

9.3.2 BIFS Command Syntax

9.3.2.1 Command Frame

```
class CommandFrame(BIFSConfig cfg) {
   do {
      Command command(cfg);
         // this is the code of a complete update command
      bit(1) continue;
   } while (continue);
}
```

9.3.2.2 Command

```
class Command(BIFSConfig cfg) {
   bit(2) code;      // this is the code of the basic Command
   switch (code) {
   case 0:
      InsertionCommand insert(cfg);
      break;

   case 1:
      DeletionCommand delete(cfg);
      break;

   case 2:
      ReplacementCommand replace(cfg);
      break;

   case 3:
      SceneReplaceCommand sceneReplace(cfg);
      break;
}
```

9.3.2.3 Insertion Command

```
class InsertionCommand(BIFSConfig cfg) {
   bit(2) parameterType ;  // this is the code of the basic Command
   switch parameterType {
   case 0:
      NodeInsertion nodeInsert(cfg);
      break;

   case 2:
      IndexedValueInsertion idxInsert(cfg);
      break;

   case 3:
      ROUTEInsertion ROUTEInsert(cfg);
      break ;
   }
}
```

9.3.2.3.1 Node Insertion

```
class NodeInsertion(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID ;
                    // this is the ID of the grouping node to which
                    // the node is added
   int ndt=GetNDTFromID(nodeID); // the node coding table of the parent
   bit(2) insertionPosition;  // the position in the children field
   switch (insertionPosition) {
   case 0:              // insertion at a specified position
      bit (8) position; // the position in the children field
      SFNode node(ndt); // the node to be inserted
      break;

   case 2:              // insertion at the beginning of the field
      SFNode node(ndt); // the node to be inserted
      break;

   case 3:              // insertion at the end of the field
      SFNode node(ndt); // the node to be inserted
      break;
   }
}
```

9.3.2.3.2 IndexedValue Insertion

```
class IndexedValueInsertion(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID;   // this is the ID of the node to be modified
   NCT nct=GetNCTFromID(nodeID); // get the NCT for this node
   int(nct.nINbits) id;          // the field to be changed
   bit(2) insertionPosition;     // the position of the value in the field
   switch (insertionPosition) {
   case 0:                       // insertion at a specified position
      bit (16) position;         // the absolute position in the field
      SFField value(nct.fieldType[id]); // the value to be inserted
      break;

   case 2:                       // insertion at the beginning of the field
      SFField value(nct.fieldType[id]);   // the value to be inserted
      break;

   case 3:                       // insertion at the end of the field
      SFField value(nct.fieldType[id]); // the value to be inserted
      break;
   }
}
```

9.3.2.3.3 ROUTE Insertion

```
class ROUTEInsertion(BIFSConfig cfg) {
   bit(1) isUpdatable;
   if (isUpdatable)
      bit(cfg.routeIDbits) routeID;
   bit(cfg.nodeIDbits) departureNodeID;   // the ID of the departure node
   NCT nctOUT=GetNCTFromID(departureNodeID);
   int(nctOUT.nOUTbits) departureID;      // the index of the departure field
   bit(cfg.nodeIDbits) arrivalNodeID;     // the ID of the arrival node
   NCT nctIN=GetNCTFromID(arrivalNodeID);
   int(nctIN.nINbits) arrivalID;          // the index of the arrival field
}
```

9.3.2.4 Deletion Command

```
class DeletionCommand(BIFSConfig cfg) {
   bit(2) parameterType ;  // this is the code of the basic Command
   switch (parameterType) {
   case 0:
      NodeDeletion nodeDelete(cfg);
      break ;

   case 2:
      IndexedValueDeletion idxDelete(cfg);
      break ;

   case 3:
      ROUTEDeletion ROUTEDelete(cfg);
      break ;
   }
}
```

9.3.2.4.1 Node Deletion

```
class NodeDeletion(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID;   // the ID of the node to be deleted
}
```

9.3.2.4.2 IndexedValue Deletion

```
class IndexedValueDeletion(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID;   // the ID of the node to be modified
   NCT nct=GetNCTFromID(nodeID);
   int(nct.nOUTbits) id;         // the field to be deleted.
   bit(2) deletionPosition;      // The position in the children field
   switch (deletionPosition) {
   case 0:                       // deletion at a specified position
      bit(16) position;          // the absolute position in the field
      SFField value(nct.fieldType[id]);   // the new value
      break;

   case 2:                       // deletion at the beginning of the field
      SFField value(nct.fieldType[id]); // the new value
      break;

   case 3:                       // deletion at the end of the field
      SFField value(nct.fieldType[id]); // the new value
      break;
   }
}
```

9.3.2.4.3 ROUTE Deletion

```
class ROUTEDeletion(BIFSConfig cfg) {
   bit(cfg.routeIDbits) routeID ;   // the ID of the ROUTE to be deleted
```

```
}
```

9.3.2.5 Replacement Command

```
class ReplacementCommand(BIFSConfig cfg) {
   bit(2) parameterType ;  // this is the code of the basic Command
   switch (parameterType) {
   case 0:
      NodeReplacement nodeReplace(cfg);
      break;

   case 1:
      FieldReplacement fieldReplace(cfg);
      break;

   case 2:
      IndexedValueReplacement idxReplace(cfg);
      break ;

   case 3:
      ROUTEReplacement ROUTEReplace(cfg);
      break;
   }
}
```

9.3.2.5.1 Node Replacement

```
class NodeReplacement(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID;   // the ID of the node to be replaced
   SFNode node(SFWorldNode);     // the new node
}
```

9.3.2.5.2 Field Replacement

```
class FieldReplacement(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID ;  // the ID of the node to be modified
   NCT nct = GetNCTFromID(nodeID);
   int(nct.nDEFbits) id;         // the index of the field to be replaced
   Field value(nct.fieldType[id]);
                                 // the new field value, either single or list
}
```

9.3.2.5.3 IndexedValue Replacement

```
class IndexedValueReplacement(BIFSConfig cfg) {
   bit(cfg.nodeIDbits) nodeID;   // this is the ID of the node to be modified
   NCT nct=GetNCTFromID(nodeID);
   int(nct.nDEFbits) id;         // the index of the field to be replaced
   bit(2) replacementPosition;   // The position in the children field
   switch (replacementPosition) {
   case 0:                       // replacement at a specified position
      bit (16) position;         // the absolute position in the field
      SFField value(nct.fieldType[id]); // the new value
      break;

   case 2:                       // replacement at the beginning of the field
      SFField value(nct.fieldType[id]); // the new value
      break;

   case 3:                       // replacement at the end of the field
      SFField value(nct.fieldType[id]); // the new value
      break;
   }
}
```

9.3.2.5.4 ROUTE Replacement

```
class ROUTEReplacement(BIFSConfig cfg) {
   bit(cfg.routeIDbits) routeID; // the ID of the ROUTE to be replaced
   bit(cfg.nodeIDbits) departureNodeID; // the ID of the departure node
   NCT nctOUT=GetNCTFromID(departureNodeID);
   int(nctOUT.nOUTbifs) departureID;   // the index of the departure field
   bit(cfg.nodeIDbits) arrivalNodeID;  // the ID of the arrival node
   NCT nctIN = GetNCTFromID(arrivalNodeID);
   int(nctIN.nINbits) arrivalID; // the index of the arrival field
}
```

9.3.2.5.5 Scene Replacement

```
class SceneReplaceCommand(BIFSConfig cfg) {
   BIFSScene scene(cfg);
      // the current scene graph is completely replaced by this new node
      // which should contain a new scene graph
}
```

9.3.3 BIFS-Anim Syntax

9.3.3.1 BIFS AnimationMask

9.3.3.1.1 AnimationMask

```
class AnimationMask(int nodeIDbits) {
   int numNodes = 0;
   do {
      ElementaryMask elemMask(nodeIDbits);
      numNodes++;
      bit(1) moreMasks;
   } while (moreMasks);
}
```

9.3.3.1.2 Elementary mask

```
Class ElementaryMask(int nodeIDbits) {
   bit(nodeIDbits) nodeID;    // The nodeID of the node to be animated
   NCT nct = getNCTFromID(nodeID);
   eMask nodeMask = buildeMaskFromNCT(nct);

   switch (NodeType) {
   case FaceType:
      // no initial mask for face
      break;

   case BodyType:
      // no initial mask for body
      break;

   case IndexedFaceSet2DType:
      break;

   default:
      InitialFieldsMask initMask(nodeMask);
            // describes which of the dynamic fields are animated and
            // their parameters
   }
}
```

9.3.3.1.3 InitialFieldsMask

```
class InitialFieldsMask(eMask nodeMask) {
   bit(1) animatedFields[nodeMask.numDYNfields];
```

```
      // read binary mask of dynamic fields for this type of node
      // numDYNfields is the number of dynamic fields of the animated node
      // the value is 1 if the field is animated, 0 otherwise
   int i;
   for(i=0; i<nodeMask.numDYNfields; i++) {
      if (animatedFields[i]) {   // is this field animated ?
         if (!isSF(nodeMask.fieldType[i]) {
            // do we have a multiple field ?
            bit(1) isTotal; // if 1, all the elements of the mfield are animated
            if (! isTotal) { // animate specified indices
            do {
               int(32) index;
               bit(1) moreIndices;
            } while (moreIndices);
         }
         InitialAnimQP QP[i](nodeMask, i);
            // read initial quantization parameters QP[i] for field i
      }
   }
}
```

9.3.3.1.4 *InitialAnimQP*

```
InitialAnimQP(eMask nodeMask, int fieldRef) {
   switch (nodeMask.animType[fieldRef])
   case 0:        // Position 3D
      float(32)   Ix3Min;
      float(32)   Iy3Min;
      float(32)   Iz3Min;
      float(32)   Ix3Max;
      float(32)   Iy3Max;
      float(32)   Iz3Max;
      float(32)   Px3Min;
      float(32)   Py3Min;
      float(32)   Pz3Min;
      float(32)   Px3Max;
      float(32)   Py3Max;
      float(32)   Pz3Max;
      int(5)      position3DNbBits;
      break;

   case 1:        // Position 2D
      float(32)   Ix2Min;
      float(32)   Iy2Min;
      float(32)   Ix2Max;
      float(32)   Iy2Max;
      float(32)   Px2Min;
      float(32)   Py2Min;
      float(32)   Px2Max;
      float(32)   Py2Max;
      int(5)      position2DNbBits;
      break;

   case 2:        // SFColor
      float(32)   IcMin;
      float(32)   IcMax;
      float(32)   PcMin;
      float(32)   PcMax;
      int(5)      colorNbBits;
      break;

   case 3:           // Angles
      float(32)   IangleMin;
      float(32)   PangleMax;
      float(32)   IangleMin;
      float(32)   PangleMax;
```

```
      int(5)      angleNbBits;
      break;

   case 4:           // Normals
      int(5)      normalNbBits;
      break;

   case 5:           // Scale
      float(32)   IscaleMin;
      float(32)   IscaleMax;
      float(32)   PscaleMin;
      float(32)   PscaleMax;
      int(5)      scaleNbBits;
      break;

   case 6:           // Rotation
      bit(1)   hasAxis;
      if (hasAxis)
         int(5)   axisNbBits;
      bit(1) hasAngle;
      if (hasAngle) {
         float(32)   IangleMin;
         float(32)   IangleMax;
         float(32)   PangleMin;
         float(32)   PangleMax;
         int(5)      angleNbBits;
      }
      break ;

   case 7:           // Size of objects
      float(32)   IsizeMin;
      float(32)   IsizeMax;
      float(32)   PsizeMin;
      float(32)   PsizeMax;
      int(5)      sizeNbBits;
      break;
   }
}
```

9.3.3.2 BIFS-Anim Frame Syntax

9.3.3.2.1 AnimationFrame

```
class AnimationFrame(BIFSConfig cfg) {
   AnimationFrameHeader header(cfg.animMask);
   AnimationFrameData data(cfg.animMask);
}
```

9.3.3.2.2 AnimationFrameHeader

```
class AnimationFrameHeader(AnimationMask mask) {
   bit(23)* next;    // read-ahead 23 bits
   if (next==0)
      bit(32) AnimationStartCode;   // synchronization code for Intra

   bit(1) isIntra;
   bit(1) animNodes[mask.numNodes]; // mask for setting which of the nodes
                                    // are animated
   if (isIntra) {
      bit(1) isFrameRate;
      if (isFrameRate)
         FrameRate rate;
      bit(1) isTimeCode;
      if (isTimeCode)
         unsigned int(18) timeCode;
```

```
   }
   bit(1) hasSkipFrames;
   if (hasSkipFrames)
      SkipFrames skip;
}

class FrameRate {
   unsigned int(8) frameRate;
   unsigned int(4) seconds;
   bit(1) frequencyOffset;
}

class SkipFrame {
   int nFrame = 0;
   do {
      bit(4) n;
      nFrame = n + nFrame*16;
   } while (n == 0b1111);
```

Kommentar: Ganesh's version from Julien's draft had here "number of frames to skip == 0b1111". I put 'n' here just to have something that looks valid, but this needs to be checked.

9.3.3.2.3 *AnimationFrameData*

```
class AnimationFrameData (AnimationMask mask, bit animNodes[mask.numNodes]) {
   int i;
   for (i=0; i<mask.numNodes; i++) {   // for each animated node
      if (animNodes[i]) {              // do we have values for node i ?
         switch (mask.nodeMask[I].nodeType) {
         case FaceType:
            FaceFrameData fdata;       // as defined in Part 2
            break;
         case BodyType:
            BodyFrameData bdata;       // as defined in Part 2
            break;

         case IndexedFaceSet2DType:
            Mesh2DframeData mdata;     // as defined in Part 2
            break;

         default:    // all other types of nodes
            int j;
            for(j=0; j<mask.nodeMask[i].numFields; j++) {
               AnimationField AField(mask.nodeMask[i], j);
                  // the syntax of the animated field; this depends
                  // on the field and node type
            }
         }
      }
   }
```

9.3.3.2.4 *AnimationField*

```
class AnimationField(eMask nodeMask, int fieldRef) {
   if (isIntra()) {
         bit(1) hasQP; // do we send new quantizationparameters ?
         if(hasQP) {
            // read new QP for field of the curent node
            AnimQP QP(nodeMask, fieldRef);
         }
         int i;
         for (i=0; i<nodeMask.numElements[fieldRef]; i++)
            AnumIValue ivalue(nodeMask, fieldRef);
                  // read intra-coded value of field
      }
      else {
         int i;
         for (i=0; i<nodeMask.numElements[fieldRef]; i++)
```

```
            AnumPValue pvalue(nodeMask, fieldRef);
                  // read predictively -coded value of field
      }
}
```

9.3.3.2.5 AnimQP

```
class AnimQP(eMask nodeMask, int fieldRef) {
   switch (nodeMask.animType[fieldRef])
   case 0:        // Position 3D
      bit (1) IMinMax ;
      if (IMinMax) {
         float(32)   Ix3Min;
         float(32)   Iy3Min;
         float(32)   Iz3Min;
         float(32)   Ix3Max;
         float(32)   Iy3Max ;
         float(32)   Iz3Max;
      }
      bit (1) PMinMax ;
      if (PMinMax) {
         float(32)   Px3Min;
         float(32)   Py3Min;
         float(32)   Pz3Min;
         float(32)   Px3Max;
         float(32)   Py3Max ;
         float(32)   Pz3Max;
      }
      bit(1) hasNbBits ;
      if (hasNbBits)
         unsigned int(5)   position3DNbBits ;
      break ;

   case 1:     // Position 2D
      bit (1) IMinMax ;
      if (IMinMax) {
         float(32)   Ix2Min;
         float(32)   Iy2Min;
         float(32)   Ix2Max;
         float(32)   Iy2Max;
      }
      bit(1) PMinMax ;
      if (PMinMax) {
         float(32)   Px2Min;
         float(32)   Py2Min;
         float(32)   Px2Max;
         float(32)   Py2Max;
      }
      bit (1) hasNbBits;
      if (hasNbBits)
         unsigned int(5) position2DNbBits;
      break ;

   case 2:        // SFColor
      bit (1) IMinMax;
      if (IMinMax) {
         float(32)   IcMin;
         float(32)   IcMax;
      }
      if (PMinMax) {
         float(32)   PcMin;
         float(32)   PcMax;
      }
      bit(1) hasNbBits;
      if (hasNbBits)
         unsigned int(5) colorNbBits;
```

```
      break ;

   case 3:           // Angles
      bit(1) IMinMax;
      if (IMinMax) {
         float(32)   IangleMin;
         float(32)   IangleMax;
      }
      bit(1) PMinMax;
      if (PMinMax) {
         float(32)   PangleMin;
         float(32)   PangleMax;
      }
      bit (1) hasNbBits;
      if (hasNbBits)
         unsigned int(5) NbBits ;
      break;

   case 4:           // Normals
      unsigned int(5)   normalNbBits ;
      break;

   case 5:           // Scale
      bit(1) IMinMax ;
      if (IMinMax) {
         float(32)   IscaleMin;
         float(32)   IscaleMax;
      }
      bit(1) PMinMax ;
      if (PMinMax) {
         float(32)   PscaleMin ;
         float(32)   PscaleMax ;
      }
      bit(1) hasNbBits ;
      if (hasNbBits)
         unsigned int(5) scaleNbBits ;
      break;

   case 6:           // Rotation
      bit(1) hasAngle;
      bit(1) hasNormal;
      bit (1) IMinMax ;
      if (IMinMax) {
         float(32)   IangleMin;
         float(32)   IangleMax;
      }
      bit(1) PMinMax;
      if (PMinMax) {
         float(32)   PangleMin ;
         float(32)   PangleMax ;
      }
      bit (1) hasNbBits ;
      if (hasNbBits) {
         if (hasAngle)
            int(5)   angleNbBits ;
         if (hasNormal)
            int(5)   normalNbBits ;
      }
      break ;

   case 7:           // Scalar
      bit (1) IMinMax ;
      if (IMinMax) {
         float(32)   IscalarMin ;
         float(32)   IscalarMax ;
      }
```

```
      bit(1) PMinMax ;
      if (PMinMax) {
         float(32)   PscalarMin ;
         float(32)   PscalarMax ;
      }
      bit (1) hasNbBits ;
      if (hasNbBits)
         unsigned int(5) scalarNbBits ;
      break;
   }
}
```

9.3.3.2.6 AnimationIValue

```
class AnimationIValue(eMask nodeMask, int fieldRef) {
   switch (nodeMask.animType[fieldRef]) {
   case 0:  // 3D Positions
      int(vlcNbBits) x;
      int(vlcNbBits) y;
      int(vlcNbBits) z;
      break;

   case 1:  // 2D Positions
      int(vlcNbBits) x;
      int(vlcNbBits) y;
      break;

   case 2:  // Color
      int(vlcNbBits) r;
      int(vlcNbBits) g;
      int(vlcNbBits) b;
      break ;

   case 3:  // Angle
      int(vlcNbBits) angle;
      break;

   case 5:  // Scale
      switch (nodeMask.fieldType[fieldRef]) {

      case SFVec2fType:
         int(vlcNbBits) xScale;
         int(vlcNbBits) yScale;
         break;

      case SFVec3fType:
         int(vlcNbBits) xScale;
         int(vlcNbBits) yScale;
         int(vlcNbBits) zScale;
         break;
      }
      break;

   case 6:  // Rotation
      if (nodeMask.hasAxis[fieldRef]) {
         int(3) octantNb;
         int(2) triantNb;
         int(vlcNbBits) xPos;
         int(vlcNbBits) yPos;
      }
      if (nodeMask.hasAngle[fieldRef])
         int(vlcNbBits) angle;
      break;

   case 7:  // Object Size and Scalar
      int(vlcNbBits) size;
```

Kommentar: Now, what's this vlcNbBits supposed to be? Is it a remnant of the old syntax for quantized fields or what?

```
      break ;
   }
}
```

9.3.3.2.7 AnimationPValue

```
class AnimationPsValue(eMask nodeMask, int fieldRef) {
   switch (nodeMask.quantType[fieldRef]) {
   case 0:  // 3D Positions
      int(vlcNbBits) x;
      int(vlcNbBits) y;
      int(vlcNbBits) z;
      break;

   case 1:  // 2D Positions
      int(vlcNbBits) x;
      int(vlcNbBits) y;
      break;

   case 2:  // Color
      int(vlcNbBits) r;
      int(vlcNbBits) g;
      int(vlcNbBits) b;
      break;

   case 3:  // Angle
      int(vlcNbBits) angle;
      break;

   case 5:  // Scale
      switch (nodeMask.fieldTtype[fieldRef]) {

      case SFVec2fType:
         int(vlcNbBits) xScale;
         int(vlcNbBits) yScale;
         break;

      case SFVec3fType:
         int(vlcNbBits) xScale;
         int(vlcNbBits) yScale;
         int(vlcNbBits) zScale;
         break;
      }
      break;

   case 6:  // Rotation
      if (nodeMask.hasAxis[fieldRef]) {
         int(3) octantNb;
         int(2) triantNb;
         int(vlcNbBits) xPos;
         int(vlcNbBits) yPos;
      }
      if (nodeMask.hasAngle[fieldRef])
         int(vlcNbBits) angle;
      break;

   case 7:  // Object Size and scalar
      int(vlcNbBits) size;
      break;
   }
}
```

9.4 BIFS Decoding Process and Semantics

The semantics of all BIFS-related structures is described in this clause. Coding of individual nodes and field values is very regular, and follows a depth-first order (children or sub-nodes of a node are present in the bitstream before its siblings). However, identification of nodes and fields within a BIFS tree depends on the context. Each field of a BIFS node that accepts nodes as fields can only accept a specific set of node types. This is referred to as the Node Data Type (or NDT) of that field. Each node belongs to one or more NDTs. NDT tables are provided in Annex H, Node Data Type Tables, which identify the various types and the nodes they contain. Identification of a particular node depends on the context of the NDT specified for its parent field. For example, **MediaTimeSensor** is identified by the 5-bit code 0b0000.1 when the context of the parent field is SF2DNode, whereas the 7-bit code 0b0000.110 is used in the context of a SFWorldNode field. The value 0 is always reserved for future extensions.

The syntactic description of fields is also dependent on the context of the node. In particular, fields are identified by code words that have node-dependent (but fixed) lengths. The type of the field is inferred by the code word, which is just an index into the position of that field within the node. Field codes are provided in Annex H, Node Coding Parameters, which provide for each node the codewords to be used to identify each of their fields. The value 0 is always reserved for future extensions.

All BIFS information is encapsulated into BIFS Command frames. These frames contain commands that perform a number of operations, such as insertion, deletion, or modification of scene nodes, their fields, or routes.

In what follows, the pertinent node data type (NDT) and node coding table (NCT) are assumed to be available as two classes of types `NDT` and `NCT` respectively. Their members provide access to particular values in the actual tables. Their definitions are as follows.

Node Data Type Table Parameters

`class NDT {`	
`int nbits;`	The number of bits used by this Node Data Type (this number is indicated in the last column of the first row of the Node Data Type table).
`int nodeID;`	This is the bitstring (code) that identifes the particular node within the context of this Node Data Type (e.g., for an AnimationStream node within an SF2DNode context is 0b0000.0).
`}`	

Node Coding Table Parameters

`class NCT {`	
`int nDEFbits;`	The number of bits used for DEF field codes (the width of the codewords in the 2^{nd} column of the tables).
`int nINbits;`	The number of bits used for IN field codes (the width of the codewords in the 3^{rd} column of the tables).
`int nOUTbits;`	The number of bits used for OUT field codes (the width of the codewords in the 4^{th} column of the tables).
`int nDYNbits;`	The number of bits used for DYN field codes (the width of the codewords in the 5^{th} column of the tables).
`int numDEFfields;`	The number of DEF fields available for this node (maximum DEF field code + 1).
`int fieldType[numDEFfields];`	The type of each DEF field (e.g., SFInt32Type). This is given by the semantic tables for each node.
`int quantType[numDEFfields];`	The type of quantization used for each DEF field (e.g., SFInt32). This is given by the node coding table for each node. Types refer to quantization tables in Subclause 9.4.1.9.
`int animType[numDEFfields];`	The type of animation for each DEF field. This is given by the node coding table for each node. Types refer to animation type tables in Subclause9.4.3.1.4 .
`int nbBits[numDEFfields];`	The number of bits used for each field. Only used in the case where the quantization category of the field is 13. In the NCT the number of bits comes right after the 13 (e.g 13, 16 in the NCT means category 13 with 16 bits).
`}`	

Kommentar: Reference?

Kommentar: Reference?

Local Quantization Parameter table

`class LQP {`

`boolean isQuantized[];`	Set to 1 if the corresponding field is quantized, 0 otherwise.
`float floatMin[numDEFfields];`	The minimum values for fields of type SFFloat. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFFloat. For other types of field, the value will be undetermined.
`float floatMax[numDEFfields];`	The maximum values for SFFloat fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFFloat. For other types of field, the value will be undetermined.
`int intMin[numDEFfields];`	The minimum values for SFFloat fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT Note that this data is only valid for quantizing fields of type SFInt32. For other types of field, the value will be undetermined.
`int intMax[numDEFfields];`	The maximum values for SFFloat fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFInt32. For other types of field, the value will be undetermined.
`float vec2fMin[numDEFfields][2];`	The minimum values for SFVec2f fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFVec2f. For other types of field, the value will be undetermined.
`float vec2fMax[numDEFfields][2];`	The maximum values for SFVec2f fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFVec2f. For other types of field, the value will be undetermined.
`float vec3fMax[numDEFfields][3];`	The maximum values for SFVec3f fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFVec3f.or SFColor. For other types of field, the value will be undetermined.
`float vec3fMin[numDEFfields][3];`	The minimum values for SFVec3f fields. These values are deducted from the NCT table and the QP min and max.values for the relevant quantization type, stored in the quantType in the NCT. Note that this data is only valid for quantizing fields of type SFVec3f.or SFColor. For other types of field, the value will be undetermined.
`int position3DnbBits;`	The number of bits for 3D positions, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int position2DnbBits;`	The number of bits for 2D positions, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int drawOrderNbBits;`	The number of bits for drawing order, as obtained from the relevant QuantizationParameter. For details on the context of

Kommentar: Figure out description of LPQ.isQuantized[] (Julien! …)

	quantization nodes, refer to Subclause 9.5.1.2.9.
`int colorNbBits;`	The number of bits for color, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int textureCoordinateNbBits;`	The number of bits for texture coordinate, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int angleNbBits;`	The number of bits for amgle, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int scaleNbBits;`	The number of bits for scale, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int keyNbBits;`	The number of bits for interpolator keys, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`int normalNbBits;`	The number of bits for normals, as obtained from the relevant QuantizationParameter. For details on the context of quantization nodes, refer to Subclause 9.5.1.2.9.
`}`	

In addition, we assume that the entire list of node data type and coding tables are available as arrays of such classes, named `ndt` and `nct` respectively, and indexed by the type (e.g., SFTopNode) and name (e.g., VideoObject2D) of the node. Finally, we assume that the following functions are available:

`LQP getCurrentLQP();`

`Returns the current value of the local quantization parameters.`

`NCT GetNCT(int nodeDataType, int nodeType)`

Returns the NCT for the particular node identified by the node data type `nodeDataType` and node type `nodeType`.

`NCT GetNCTFromID(int id)`

Returns the NCT for the particular node identified by the node `id`.

`int isSF(int fieldType)`

Returns 1 if the `fieldType` corresponds to an SF* field (e.g., SFInt32) and 0 otherwise (e.g., MFInt32).

`int GetNDTFromID(int id)`

Returns the node data type for the `node` field of the particular node identified by the node `id`.

9.4.1 BIFS Scene and Nodes Decoding Process

The scene syntax describes the syntax for an entire BIFS scene description. However, this syntax is also used in combination with a BIFS-Command, as described in the next section.

9.4.1.1 BIFS Decoder Configuration

BIFSConfig is the decoder configuration for the BIFS elementary stream. It is encapsulated within the 'specificInfo' fields of the general **DecoderSpecificInfo** structure (Subclause 8.3.5), which is contained in the **DecoderConfigDescriptor** that is carried in **ES_Descriptors**.

The parameter **nodeIDbits** sets the number of bits used to represent node IDs. Similarly, **routeIDbits** sets the number of bits used to represent ROUTE IDs.

The boolean **isCommandStream** identifies whether the BIFS stream is a BIFS-Command stream or a BIFS-Anim stream. If the BIFS-Command stream is selected (**isCommandStream** is set to 1), the following parameters are sent in the bitstream.

The boolean **isPixelMetric** indicates whether pixel metrics or meter metrics are used.

The boolean **hasSize** indicates whether a desired scene size (in pixels) is specified. If **hasSize** is set to 1, **pixelWidth** and **pixelHeight** provide to the receiving terminal the desired horizontal and vertical dimensions (in pixels) of the scene.

If **isCommandStream** is 0, a BIFS-Anim elementary stream is expected and the **AnimationMask** is sent for setting the relevant animation parameters.

9.4.1.2 BIFS Scene

The BIFSScene structure represents the global scene. A BIFSScene is always associated to a ReplaceScene BIFS-Command message. The BIFSScene is structured in the following way:

- the nodes of the scene are described first, and
- ROUTEs are described after all nodes

The scene always starts with a node of type SFTopNode. This implies that the top node can be either a **Group2D** , **Layer2D**, **Group** or **Layer3D** node.

9.4.1.3 SFNode

The **SFNode** represents a generic node. The encoding depends on the Node Data Type of the node (NDT).

If **isReused** true ('1') then this node is a reference to another node, identified by its **nodeID**. This is similar to the use of the **USE** statement in ISO/IEC 14772-1.

If **isReused** is false ('0'), then a complete node is provided in the bitstream. This requires that the **NodeType** is inferred from the Node Data Type, as explained in the following section.

The **isUpdatable** flag enables the assignment of a **nodeID** to the node. This is equivalent to the **DEF** statement of ISO/IEC 14772-1.

The node definition is then sent, either with a **MaskNodeDescription**, or a **ListNodeDescription**.

9.4.1.4 MaskNodeDescription

In the **MaskNodeDescription**, a **Mask** indicates for each field of type DEF of the node (according to the node type), if the field value is provided. Fields are sent in the order indicated in Annex H, Node Coding Parameters. According to the order of the fields, the field type is known and is used to decode the field.

9.4.1.5 ListNodeDescription

In the **ListNodeDescription**, fields are directly addressed by their field reference, **fieldRef**. The reference is sent as a **defID** and its parsing depends on the node type, as explained in the defID section.

9.4.1.6 NodeType

The **nodeType** is a number that represents the type of the node. This **nodeType** is coded using a variable number of bits for efficiency reasons. The following explains how to determine the exact type of node from the nodeType:

1. The data type of the field parsed indicates the Node Data Type: **SF2DNode**, **SFColorNode**, and so on. The root node is always of type **SFTopNode**.
2. From the Node Data Type expected and the total number of nodes type in the category, the number of bits representing the **nodeType** is obtained (this number is shown in the Node Data Type tables in Annex H).
3. Finally, the **nodeType** gives the nature of the node to be parsed.

Example:

The **Shape** node has 2 fields defined as:

```
exposedField SFAppearanceNode Appearance NULL
exposedField SFGeometry3DNode   geometry   NULL
```

When decoding a **Shape** node, if the first field is transmitted, a node of type **SFAppearanceNode** is expected. The only node with **SFAppearanceNode** type is the **Appearance** node, and hence the **nodeType**can be coded using 0 bits. When decoding the **Appearance** node, the following fields can be found:

```
exposedField SFMaterialNode          Material          NULL
exposedField SFTextureNode           texture           NULL
exposedField SFTextureTransformNode TextureTransform  NULL
```

If a texture is applied on the geometry, a texture field will be transmitted. Currently, the MovieTexture, the PixelTexture and ImageTexture, Composite2Dtexture and Composite3DTexture are available. This means that the **nodeType** for the texture node can be coded using 3 bits.

9.4.1.7 Field

A **field** is encoded according to its type: single (SField) or multiple (MField). A multiple field is a collection of single fields.

9.4.1.8 MFField

MFField types can be encoded with a list (**ListFieldDescription**) or mask (**MaskFieldDescription**) description. The choice depends on the encoder, and is normally made according to the number of elements in the multiple field.

9.4.1.9 SFField

Single fields are coded according to the type of the field. The fields have a default syntax that specifies a raw encoding when no quantization is applied. The semantics of the fields are self-explanatory.

For floating point values it is possible to use a more economical representation than the standard 32-bit format, as specified in the **Float** structure. This representation separately encodes the size of the exponent (base 2) and mantissa of the number, so that less than 32 bits may be used for typically used numbers. Note that when the **mantissaLength** is 0, the **value** is 0. Also, if the **exponentLength** is 0 then the **exponent** is not parsed, and – by default – assumes the value 0.

When quantization is used, the quantization parameters are obtained from a special node called **QuantizationParameter**. The following quantization categories are specified, providing suitable quantization procedures for the various types of quantities represented by the various fields of the BIFS nodees.

Table 9-2: Quantization Categories

Category	Description
0	None
1	3D Position
2	2D positions
3	depth
4	SFColor
5	Texture Coordinate
6	Angle
7	Scale
8	Interpolator keys
9	Normals
10	Rotations (9+5)
11	Object Size 3D (1)
12	Object Size 2D (2)
13	Linear Scalar Quantization
14	Reserved

Each field that may be quantized is assigned to one of the quantization categories (see Node Coding Tables, Annex H). Along with quantization parameters, minimum and maximum values are specified for each field of each node.

The scope of quantization is only a single BIFS access unit.

A field is quantized under the following conditions:

1. The field is of type SFInt32, SFFloat, SFRotation, SFVec2f or SFVec3f
2. The quantization category of the field is not 0.
3. The node to which the field belongs has a QuantizationParameter node in its context
4. The quantization for the quantization category of the field is activated (by setting the corresponding boolean to 'true' in the QuantizationParameter of the node.

9.4.1.10 QuantizedField

The decoding of fields operates using the LQP and NCT tables. The basic principle is to obtain from the NCT and the QuantizationParameter node in the scope of the node the quantization information and then to perform a linear dequantization of the data. The basic information is the minimum value, the maximum value and the number of bits obtained. In the following clauses, the way to obtain conditions for a field to be quantized, then the minimum and maximum values from the NCT and the QuantizationParameter node is described. Finally, the quantization methods applied according to the field type are described.

The following rules are applied to fill in the local quantization parameter table.

9.4.1.10.1 Local Quantization Parameter Table

`isQuantized[]`

For each field (indexed by fieldRef), isQuantized is set to true when:

```
nct.quantType[fieldRef] !=0
```

and the following condition is met for the relevant quantization type:

Quantization Type nct.quantType[fieldRef]	Condition
1	`qp.position3Dquant == TRUE`
2	`qp.position2Dquant == TRUE`
3	`qp.drawOrderQuant == TRUE`
4	`qp.colorQuant == TRUE`
5	`qp.textureCoordinateQuant == TRUE`
6	`qp.angleQuant == TRUE`
7	`qp.scaleQuant == TRUE`
8	`qp.keyQuant == TRUE`
9	`qp.normalQuant == TRUE`
10	`qp.normalQuant == TRUE`
11	`qp.position3Dquant == TRUE`
12	`qp.position2Dquant == TRUE`
13	`always TRUE`
14	`always TRUE`

`floatMax[]`
For each field (indexed by fieldRef), floatMax is set in the following way:

Quantization type nct.quantType[fieldRef]	floatMax[fieldRef]
1	`0.0.`
2	`0.0`
3	`min(nct.min[fieldRef],qp.drawOrderMax)`
4	`0.0`
5	
6	`min(nct.min[fieldRef],qp.angleMax)`
7	`0.0`

8	min(nct.min[fieldRef],qp.keyMax)
9	0.0
10	0.0
11	$\lVert \mathrm{qp}.\mathit{position}2\mathrm{DMax} - \mathrm{qp}.\mathit{position}2\mathrm{DMin} \rVert_2$
12	$\lVert \mathrm{qp}.\mathit{position}3\mathrm{DMax} - \mathrm{qp}.\mathit{position}3\mathit{DMin} \rVert_2$
13	$2^{\mathrm{nct.nbBits[fieldRef]}}-1$
14	0.0

floatMin[]

For each field (indexed by fieldRef), floatMin is set in the following way:

Quantization type nct.quantType[fieldRef]	**floatMin[fieldRef]**
1	0.0.
2	0.0
3	min(nct.min[fieldRef],qp.drawOrderNbBits)
4	0.0
5	
6	min(nct.min[fieldRef],qp.angleNbBits)
7	0.0
8	min(nct.min[fieldRef],qp.keyNbBits)
9	0.0
10	0.0
11	0.0
12	0.0
13	0.0
14	0.0

intMax[]

For each field (indexed by fieldRef), intMax is set in the following way:

Quantization type nct.quantType[fieldRef]	**intMax[fieldRef]**
1	0
2	0
3	min(nct.min[fieldRef],qp.drawOrderNbBits)
4	0
5	0
6	min(nct.min[fieldRef],qp.angleNbBits)
7	0
8	min(nct.min[fieldRef],qp.keyNbBits)
9	0
10	0
11	$\lVert \mathrm{qp}.\mathit{position}2\mathrm{DMax} - \mathrm{qp}.\mathit{position}2\mathrm{DMin} \rVert_2$
12	$\lVert \mathrm{qp}.\mathit{position}3\mathrm{DMax} - \mathrm{qp}.\mathit{position}3\mathit{DMin} \rVert_2$
13	$2^{\mathrm{nct.nbBits[fieldRef]}}-1$
14	0

intMin[]

For each field (indexed by fieldRef), floatMin is set in the following way:

Quantization type nct.quantType[fieldRef]	intMin[fieldRef]
1	0
2	0
3	min(nct.min[fieldRef],qp.drawOrderNbBits)
4	0
5	0
6	min(nct.min[fieldRef],qp.angleNbBits)
7	0
8	min(nct.min[fieldRef],qp.keyNbBits)
9	0
10	0
11	0
12	0
13	0
14	0

vec2fMax[]

For each field (indexed by fieldRef), vec2fMax is set in the following way:

Quantization type nct.quantType[fieldRef]	vec2fMax[fieldRef]
1	0.0,0.0.
2	qp.position2Dmax
3	0.0,0.0
4	0.0,0.0
5	0.0,0.0
6	0.0,0.0
7	0.0,0.0
8	0.0,0.0
9	0.0,0.0
10	0.0,0.0
11	0.0,0.0
12	$\left\Vert \mathrm{qp}.position2\mathrm{DMax} - \mathrm{qp}.position2\mathrm{DMin} \right\Vert_2$, $\left\Vert \mathrm{qp}.position2\mathrm{DMax} - \mathrm{qp}.position2\mathrm{DMin} \right\Vert_2$
13	$\left\Vert \mathrm{qp}.position3\mathrm{DMax} - \mathrm{qp}.position3DMin \right\Vert_2$, $\left\Vert \mathrm{qp}.position3\mathrm{DMax} - \mathrm{qp}.position3DMin \right\Vert_2$
14	$2^{\texttt{nct.nbBits[fieldRef]}}-1$

vec2fMin[]

For each field (indexed by fieldRef), vec2fMin is set in the following way:

Quantization type nct.quantType[fieldRef]	vec2fMin[fieldRef]
1	0.0, 0.0
2	qp.position2Dmin
3	0.0, 0.0
4	0.0, 0.0
5	0.0, 0.0
6	0.0, 0.0
7	0.0, 0.0

8	0.0, 0.0
9	0.0, 0.0
10	0.0, 0.0
11	0.0, 0.0
12	0.0, 0.0
13	0.0, 0.0
14	0.0, 0.0

vec3fMax[]

For each field (indexed by fieldRef), vec3fMax is set in the following way:

Quantization type nct.quantType[fieldRef]	**vec3fMax[fieldRef]**
1	qp.position3Dmax.
2	0.0, 0.0, 0.0
3	qp.colorMax, qp.colorMax, qp.colorMax
4	0.0, 0.0, 0.0
5	0.0, 0.0, 0.0
6	0.0, 0.0, 0.0
7	qp.scaleMax, qp.scaleMax, qp.scaleMax
8	0.0, 0.0, 0.0
9	0.0, 0.0, 0.0
10	0.0, 0.0, 0.0
11	0.0, 0.0, 0.0
12	$\lVert \mathrm{qp.}\mathit{position}2\mathrm{DMax} - \mathrm{qp.}\mathit{position}2\mathrm{DMin} \rVert_2$, $\lVert \mathrm{qp.}\mathit{position}2\mathrm{DMax} - \mathrm{qp.}\mathit{position}2\mathrm{DMin} \rVert_2$, $\lVert \mathrm{qp.}\mathit{position}2\mathrm{DMax} - \mathrm{qp.}\mathit{position}2\mathrm{DMin} \rVert_2$
13	$\lVert \mathrm{qp.}\mathit{position}3\mathrm{DMax} - \mathrm{qp.}\mathit{position}3\mathit{DMin} \rVert_2$, $\lVert \mathrm{qp.}\mathit{position}2\mathrm{DMax} - \mathrm{qp.}\mathit{position}2\mathrm{DMin} \rVert_2$, $\lVert \mathrm{qp.}\mathit{position}2\mathrm{DMax} - \mathrm{qp.}\mathit{position}2\mathrm{DMin} \rVert_2$
14	$2^{\texttt{nct.nbBits[fieldRef]}}-1$

vec3fMin[]

For each field (indexed by fieldRef), vec3fMin is set in the following way:

Quantization type nct.quantType[fieldRef]	**vec3fMin[fieldRef]**
1	qp.position3DMin
2	0.0, 0.0, 0.0
3	qp.colorMin, qp.colorMax, qp.colorMin
4	0.0, 0.0, 0.0
5	0.0, 0.0, 0.0
6	0.0, 0.0, 0.0
7	qp.scaleMin, qp.scaleMax, qp.scaleMin
8	0.0, 0.0, 0.0
9	0.0, 0.0, 0.0
10	0.0, 0.0, 0.0
11	0.0, 0.0, 0.0
12	0.0, 0.0, 0.0
13	0.0, 0.0, 0.0
14	0.0, 0.0, 0.0

9.4.1.10.2 Quantization method

For each of the field types, the following formulas are used for inverse quantizatio. $\hat{v}$ represents the inverse quantized value, v_q the quantized value, v the original value, and N the number of bits used (as defined in the syntax specification).

Field Type	**Quantization/Inverse Quantization Method**
SFInt32	For fields of type SFInt32, the quantized value is the integer shifted to fit the interval $[0, 2^N -1]$. $v_q = v - \text{lpq.intMin[fieldRef]}$ $\hat{v} = \text{lpq.intMin[fieldRef]} + v_q$
SFImage	• For SFImage types, the width and height of the image are sent. **numComponents** defines the image type. The 4 following types are enabled: - Ifthe value is ‘00’, then a grey scale image is defined. - If the value is ‘01’, a grey scale with alpha channel is used. - If the value is ‘10’, then an r, g, b image is used. - If the value is ‘11’, then an r,g, b image with alpha channel is used.
SFFloat	$v_q = \frac{v - \mathit{lpq.floatMin[fieldRef]}}{\mathit{lpq.floatMax[fieldRef]}\text{-}\mathit{lpq.floatMin[fieldRef]}}(2^N - 1)$ For the inverse quantization: $\hat{v} = \mathit{lpq.floatMin[fieldRef]} + \frac{\mathit{lpq.floatMax[fieldRef]}\text{-}\mathit{lpq.floatMin[fieldRef]}}{2^N - 1} v_q$
SFRotation	• fields of type **SFRotation** are made of 4 floats: 3 for an axis of rotation and 1 for an angle. For this field, two quantizers are used: one for the axis of rotation which is quantized as a normal (see below) and one for the angle, which is quantized as a float.
SFVec2f	For each component of the vector, the float quantization is applied: $v_q[i] = \frac{v[i] - \mathit{lpq.floatMin[fieldRef]}[i]}{\mathit{lpq.floatMax[fieldRef]}[i]\text{-}\mathit{lpq.floatMin[fieldRef]}[\mathrm{i}]}(2^N - 1)$ For the inverse quantization: $\hat{v}[i] = \mathit{lpq.floatMin[fieldRef]}[i] + \frac{\mathit{lpq.floatMax[fieldRef]}[i]\text{-}\mathit{lpq.floatMin[fieldRef]}[\mathrm{i}]}{2^N - 1} v_q[i]$
SFVec3f, if quantType = 9 (normals)	For normals, the quantization method is the following: A normal is a set of 3 floating values representing a vector in 3-d space with unit length. The quantization process first divides the unit sphere into eight octants. The signs of the 3 coordinates of the nomal determine the octant and the first 3 bits of the quantized normal. Then each octant is further symetrically divided into 3 ‘triants’ of equal size (a triant is a quadrilateral on the sphere). The index of the most significant coordinate (the one with the largest absolute value) determines the triant and the 2 next bits. Each triant is then mapped into a unit square. Finally each axis of the square is evenly subdivided into $2^{\text{normalNbBits}}$ so that position within a triant can be associated with a couple (a_q,b_q), where a_q and b_q have integer values between 0 and $2^{\text{normalNbBits}}$ -1 The quantization is the one of a SFVec2f with min = (0.0,0.0), max = (1.0,1.0), and N= normalNbBits.

	The mapping of the triant {x>0, y>0, z>0, x>z, x>y} into a unit square is $a = \frac{4}{\pi}\tan^{-1}\left(\frac{y}{x}\right)$, $b = \frac{4}{\pi}\tan^{-1}\left(\frac{z}{x}\right)$. The inverse mapping is $x = \frac{1}{\sqrt{1+\tan^2\frac{a\pi}{4}+\tan^2\frac{b\pi}{4}}}$, $y = \frac{\tan\frac{a\pi}{4}}{\sqrt{1+\tan^2\frac{a\pi}{4}+\tan^2\frac{b\pi}{4}}}$, $z = \frac{\tan\frac{b\pi}{4}}{\sqrt{1+\tan^2\frac{a\pi}{4}+\tan^2\frac{b\pi}{4}}}$. The mapping is defined similarly for the other triants. 3 bits are used to designate the octant used. 2 bits are used to designate the triant. The parameter normalNbBits specifies the number of bits used to quantize positions on the 2D square.
SFVec3f, other	For each component of the vector, the float quantization is applied: $v_q[i] = \frac{v[i] - lpq.floatMin[fieldRef][i]}{lpq.floatMax[fieldRef][i]\text{-}lpq.floatMin[fieldRef][i]}(2^N - 1)$ For the inverse quantization: $\hat{v}[i] = lpq.floatMin[fieldRef][i] + \frac{lpq.floatMax[fieldRef][i]\text{-}lpq.floatMin[fieldRef][i]}{2^N - 1}v_q[i]$

9.4.1.11 Field and Events IDs Decoding Process

Four different fieldIDs are defined to refer to fields in the nodes. All field IDs are encoded with a variable number of bits. For each field of each node, the binary values of the field IDs are defined in the node tables.

9.4.1.11.1 DefID

The **defID**s correspond to the IDs for the fields defined with node declarations. They include fields of type **exposedField** and **field**.

9.4.1.11.2 inID

The **inID**s correspond to the IDs for the events and fields that can be modified from outside the node. They include fields of type **exposedField** and **eventIn** types.

9.4.1.11.3 outID

The **outID**s correspond to the IDs for the events and fields that can be output from the node. They includes fields of type **exposedField** and **eventOut** types.

9.4.1.11.4 dynID

The **dynID**s correspond to the IDs for fields that can be animated using the BIFS-Anim scheme. They correspond to a subset of the fields designated by **inID**s.

9.4.1.12 ROUTE Decoding Process

ROUTEs are encoded using list or vector descriptions. Similar to nodes, ROUTEs can be assigned an ID. **inID** and **outID** are used for the ROUTE syntax.

9.4.2 BIFS-Command Decoding Process

9.4.2.1 Command Frame

A **CommandFrame** is a collection of BIFS-Commands, and corresponds to one access unit.

9.4.2.2 Command

For each **Command**, the 2-bit flag **command** signals one of the 4 basic commands: insertion, deletion, replacement, and scene replacement.

9.4.2.3 Insertion Command

There are four basic insertion commands, signaled by a 2 bit flag.

9.4.2.3.1 Node Insertion

A node can be inserted in the **children** field of a grouping node. The node can be inserted at the beginning, at the end, or at a specified position in the children list. The NDT of the inserted node is known from the NDT of the **children** field in which the node is inserted.

9.4.2.3.2 IndexedValue Insertion

The field in which the value is inserted must a multiple value type of field. The field is signaled with an **inID**. The **inID** is parsed using the table for the node type of the node in which the value is inserted, which is inferred from the **nodeID**.

9.4.2.3.3 ROUTE Insertion

A ROUTE is inserted in the list of ROUTEs simply by specifying a new ROUTE.

9.4.2.4 Deletion Command

There are three types of deletion commands, signalled by a 2-bit flag.

9.4.2.4.1 Node Deletion

The node deletion is simply signalled by the **nodeID** of the node to be deleted. When deleting a node, all fields are also deleted, as well as all **ROUTE**s related to the node or its fields.

9.4.2.4.2 IndexedValue Deletion

This command enables to delete an element of a multiple value field. As for the insertion, it is possible to delete at a specified position, at the beginning or at the end.

9.4.2.4.3 ROUTE Deletion

Deleting a ROUTE is simply performed by giving the ROUTE ID. This is similar to the deletion of a node.

9.4.2.5 Replacement Command

There are 3 replacement commands, signalled by a 2-bit flag.

9.4.2.5.1 Node Replacement

When a node is replaced, all the ROUTEs pointing to this node are deleted. The node to be replaced is signaled by its **nodeID**. The new node is encoded with the SFWorldNode Node Data Type, which is valid for all BIFS nodes, in order to avoid necessitating the NDT of the replaced node to be established .

Kommentar: You made our lifes difficult by using NDTs, and now you want to avoid the lookup?! If we are so committed in saving bits, this should be changed.

9.4.2.5.2 Field Replacement

This commands enables the replacement of a field of an existing node. The node in which the field is replaced is signaled with the **nodeID**. The field is signaled with an **inID**, which is encoded according to the node type of the changed node.

9.4.2.5.3 IndexedValue Replacement

This command enables the modification of the value of an element of a multiple field. As for any multiple field access, it is possible to replace at the beginning, the end or at a specified position in the multiple field.

9.4.2.5.4 ROUTE Replacement

Replacing a ROUTE deletes the replaced ROUTE and replaces it with the new ROUTE.

9.4.2.5.5 Scene Replacement

Replacing a new scene simply consists in replacing entirely the scene with a new BIFSScene scene. When used in the context of an Inline/Inline2D node, this corresponds to replacement of the subscene (previously assumed to be empty). Thus this simply inserts a new sub scene as expected in an Inline/Inline2D node.

In a BIFS elementary stream, the SceneReplacement commands are the only random access points.

9.4.3 BIFS-Anim Decoding Process

The dynamic fields are quantized and coded by a predictive coding scheme as shown in Figure 9-8. For each parameter to be coded in the current frame, the decoded value of this parameter in the previous frame is used as the prediction. Then the prediction error, i.e., the difference between the current parameter and its prediction, is computed and coded using entropy coding. This predictive coding scheme prevents the coding error from accumulating.

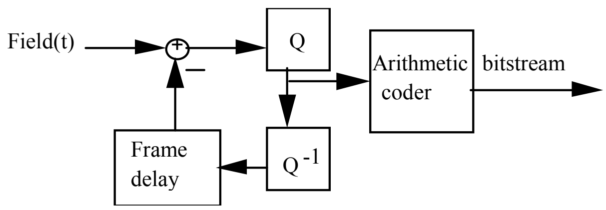


Figure 9-8: Encoding dynamic fields

The **InitialAnimQP** or **AnimQP** structures specifiy quantization parameters that enable to control the quality and precision of the reconstructed animation stream.

The decoding process performs the reverse operations, by applying first an adaptive arithmetic decoder, then the inverse quantization and adding the previous field, in predictive (P) mode, or taking the new value directly in Intra (I) mode.

Kommentar: Where is the decoding procedure specified??

The BIFS-Anim structures have two parts: the **AnimationMask**, that sets the nodes and fields to be animated, and the **AnimationFrame**, that contains the data setting new values to those fields.

The animation mask is sent in the DecoderConfigDescriptor field of the object descriptor, whereas the animation frames are sent in a separate BIFS stream. When parsing the BIFS-Anim stream, the NCT structure and related functions as described in Annex H are known. Additional structures, **aMask** and **eMask** are constructed from the **AnimationMask** and further used in the decoding process of the BIFS-Anim frames.

Kommentar: These were defined in Julien's update. However, I don't see where they are used (i.e., why the AnimationMask itself is not sufficient). At any rate, the description of the whole BIFS-Anim business has serious problems, and it is by far the worst part of the spec.

`class aMask {`	
`int numNodes;`	The number of nodes to be animated
`eMask nodeMask[numNodes];`	The elementary mask for each of the nodes being animated.
`float[3] Iposition3DMin;`	The minimum value for position 3D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[3] Iposition3DMax;`	The maximum value for position 3D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int Iposition3DNbBits;`	The number of bits for position 3D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[2] Iposition2DMin;`	The minimum value for position 2D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[2] Iposition2DMax;`	The maximum value for position 2D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int Iposition2DNbBits;`	The number of bits for position 2D in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IcolorMin;`	The minimum value for color in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IcolorMax;`	The maximum value for color in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int IcolorNbBits;`	The number of bits for color in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IangleMin;`	The minimum value for angle in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IangleMax;`	The maximum value for angle in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int IangleNbBits;`	The number of bits for angle in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int InormalNbBits;`	The number of bits for normals in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IscaleMin;`	The minimum value for scale factors and scalars in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float IscaleMax;`	The maximum value for scale factors and scalars in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int IscaleNbBits;`	The number of bits for scale factors and scalars in I mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[3] Pposition3DMin;`	The minimum value for position 3D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[3] Pposition3DMax;`	The maximum value for position 3D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int Pposition3DNbBits;`	The number of bits for position 3D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[2] Pposition2DMin;`	The minimum value for position 2D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float[2] Pposition2DMax;`	The maximum value for position 2D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int Pposition2DNbBits;`	The number of bits for position 2D in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float PcolorMin;`	The minimum value for color in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.

`float PcolorMax;`	The maximum value for color in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int PcolorNbBits;`	The number of bits for color in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float PangleMin;`	The minimum value for angle in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float PangleMax;`	The maximum value for angle in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int PangleNbBits;`	The number of bits for angle in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int PnormalNbBits;`	The number of bits for normals in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float PscalarMin;`	The minimum value for scale factors and scalars in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`float PscalarMax;`	The maximum value for scale factors and scalars in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`int PscaleNbBits;`	The number of bits for scale factors and scalars in P mode. This value is deducted from the InitialAnimQP or AnimQP structure.
`}`	

`class eMask {`	
`int nodeType;`	The node type of the node being animated.
`int numDYNfields;`	The number of bits used for IN field codes (the width of the codewords in the 3rd column of the tables).
`boolean hasAxis[numDYNfields];`	The flags indicate for fields of type SFRotation if the axis is being animated.
`boolean hasAngle[numDYNfields];`	The flags indicate for fields of type Sfotation if the angle is being animated.
`int elementList[numDYNfields][];`	The element list for SFField being animated.. For instance, if field 5 is a MFField with elements 3,4 and 7 being animated, the valuse of elementList[5] will be {3,4,7}.
`int numElements[numDYNfields];`	The number of elements being animated in a field. This is 1 for all single fields, and more than or 1 for multiple fields.
`int fieldType[numDYNfields];`	The type of each DEF field (e.g., SFInt32). This is given by the semantic tables for each node, and indexed with the DYNids.
`int animType[numDYNfields];`	The type of animation for each DEF field This is given by the semantic tables for each node. Types refer to animation type in the Node Coding Tables in Annex H. The vector is indexed with DYNids.
`}`	

Additionally, we suppose that the following functions are available:

`aMask getCurrentaMask()`

Obtain the animation mask for the BIFS session.

`void isIntra()`

Returns 1 if in Intra mode for the current decoded frame, 0 if in P mode.

9.4.3.1 BIFS AnimationMask

The **AnimationMask** sets up the parameters for an animation. In particular, it specifies the fields and the nodes to be animated in the scene and their parameters. The Mask is sent in the **ObjectDescriptor** pointing to the BIFS-Anim stream.

9.4.3.1.1 AnimationMask

The **AnimationMask** of **ElementaryMask** for animated nodes and their associated parameters.

9.4.3.1.2 Elementary mask

The **ElementaryMask** links an **InitialFieldsMask** with a node specified by its **nodeID**. The InitialFieldsMask is not used for FDP, BDP or IndexedFaceSet2D nodes.

9.4.3.1.3 InitialFieldsMask

The InitialFieldsMask specifies which fields of a given node are animated. In the case of a multiple field, either all the fields or a selected list of fields are animated.

9.4.3.1.4 InitialAnimQP

The initial quantization masks are defined according to the categories of fields addressed. In the nodes specification, it is specified for each field whether it is a dynamic field or no, and in the case which type of quantization and coding scheme is applied. The fields are grouped in the following category for animation:

Table 9-3: Animation Categories

Category	Description
0	3D Position
1	2D positions
2	Color
3	Angle
4	Normals
5	Scale
6	Rotations3D
7	Object Size

For each type of quantization, the min and max values for I and P mode, as well as the number of bits to be used for each type is specified. For the rotation, it is possible to choose to animate the angle and/or the axis with the **hasAxis** and **hasAngle** bits. When the flags are set to TRUE, the validity of the flag is for the currently decoded frame or until the next **AnimQP** that sets the flag to a different value.

9.4.3.2 Animation Frame Decoding Process

9.4.3.2.1 AnimationFrame

The **AnimationFrame** is the **Access Unit** for the BIFS-Anim stream. It contains the **AnimationFrameHeader**, which specifies some timing, and selects which nodes are being animated in the list of animated nodes, and the **AnimationFrameData**, which contains the data for all nodes being animated.

9.4.3.2.2 AnimationFrameHeader

In the **AnimationFrameHeader**, a start code is sent optionally at each I or P frame. Additionally, a mask for nodes being animated is sent. The mask has the length of the number of nodes specified in the **AnimationMask**. A ‘1’ in the header specifies that the node is animated for that frame, whereas a ‘0’ that is is not animated in the current frame. In the header, if in **Intra** mode, some additional timing information is also specified. The timing information follows the syntax of the Facial Animation specification in the Part 2 of this Final Committee Draft of International Standard.

Finally, it is possible to skip a number of **AnimationFrame** by using the **FrameSkip** syntax specified in Part 2 of this Final Committee Draft of International Standard.

9.4.3.2.3 AnimationFrameData

The **AnimationFrameData** corresponds to the field data for the nodes being animated. In the case of an IndexedFaceSet2D, a face, or a body, the syntax used is the one defined in Part 2 of this Final Committee Draft of International Standard. In other cases, for each field animated node and for each animated field the **AnimationField** is sent.

9.4.3.2.4 AnimationField

In an **AnimationField**, if in **Intra** mode, a new **QuantizationParameter** value is optionally sent. Then comes the I or P frame.

All numerical parameters as defined in the categories below follow the same coding scheme.

- In P (Predictive) mode: for each new value to send, we code its difference with the preceding value. Values are quantized with a uniform scalar scheme, and then coded with an adaptive arithmetic encoder, as described in Part 2 of this Final Committee Draft of International Standard.
- In I (Intra) mode: values of dynamic fields are directly quantized and coded with the same arithmetic adaptive coding scheme

The syntax for all the numerical field animation is the same for all types of fields. The category corresponds to Table 9-3 above.

9.4.3.2.5 AnimQP

The **AnimQP** is identical to the **InitialAnimQP**, except that it enables to send minimum and maximum values as well as the number of bits for quantization for each type of field.

9.4.3.2.6 AnimationIValue and AnimationPValue

9.4.3.2.6.1 Quantization of I Values

The quantization methods of values is similar to the quantization for BIFS scenes, except that the minimum, maximum and number of bits information is directly obtained from the animation quantization parameters (see **InitialAnimQP** and **AnimQP**), either in the animation mask, or (and) in I frames.

The following table details the qunatization and inverse quantization process for each of the categories.

For each of the field types, the following formulas are used for inverse quantization: $\hat{v}$ represents the inverse quantized value, v_q the quantized value, v the original value, N the number of bits used (as declared in the syntax specification). All the inverse quantization process always takes into account the new values of the animQP which are sent in the current I frame. When no new value is available for a specific coding parameter (number of biits, min, max..) or when no AnimQP is sent in the I frame, the values as in the **initial animQP** are applied.

Category	Quantization/Inverse Quantization Method
0	For each component of the SFVec3f vector, the float quantization is applied: $v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ For the inverse quantization: $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$
1	For each component of the SFVec2f vector, the float quantization is applied:

	$v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ For the inverse quantization: $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$
3	For the angles, the float quantization is applied: $v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ For the inverse quantization: $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$
4	For normals, the quantization method is the following: A normal is a set of 3 floating values representing a vector in 3-d space with unit length. The quantization process first divides the unit sphere into eight octants. The signs of the 3 coordinates of the nomal determine the octant and the first 3 bits of the quantized normal. Then each octant is further symetrically divided into 3 ‘triants’ of equal size (a triant is a quadrilateral on the sphere). The index of the most significant coordinate (the one with the largest absolute value) determines the triant and the 2 next bits. Each triant is then mapped into a unit square. Finally each axis of the square is evenly subdivided into $2^{normalNbBits}$ so that position within a triant can be associated with a couple (a_q,b_q), where a_q and b_q have integer values between 0 and $2^{mask.InormalNbBits}$ -1 The quantization is the one of a SFVec2f (case 1) with min = (0.0,0.0), max = (1.0,1.0), and N= normalNbBits. The mapping of the triant {x>0, y>0, z>0, x>z, x>y} into a unit square is $a = \frac{4}{\pi}\tan^{-1}\left(\frac{y}{x}\right)$, $b = \frac{4}{\pi}\tan^{-1}\left(\frac{z}{x}\right)$. The inverse mapping is $x = \frac{1}{\sqrt{1 + \tan^2\frac{a\pi}{4} + \tan^2\frac{b\pi}{4}}}$, $y = \frac{\tan\frac{a\pi}{4}}{\sqrt{1 + \tan^2\frac{a\pi}{4} + \tan^2\frac{b\pi}{4}}}$, $z = \frac{\tan\frac{b\pi}{4}}{\sqrt{1 + \tan^2\frac{a\pi}{4} + \tan^2\frac{b\pi}{4}}}$. The mapping is defined similarly for the other triants. 3 bits are used to designate the octant used. 2 bits are used to designate the triant. The parameter normalNbBits specifies the number of bits used to quantize positions on the 2D square.
5	For each component of the SFVec2f vector, the float quantization is applied: $v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ For the inverse quantization: $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$
6	fields of type **SFRotation** are made of 4 floats: 3 for an axis of rotation and 1 for an angle. For this field, two quantizers are used: one for the axis of rotation which is quantized as a normal (see below) and one

	for the angle, which is quantized as a float.
7 (SFInt32)	For fields of type SFInt32, the quantized value is the integer shifted to fit the interval [0, $2^{mask.IscaleNbBits}$ - 1]. $v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$
7 (SFFloat)	$v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$ For the inverse quantization: $\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$

9.4.3.2.6.2 Decoding Process

The decoding process in P mode computes the animation values by applying first the adaptive arithmetic decoder, then the inverse quantization and adding the previous field. Let v(t) be the value decoded at an instant t, v(t-1) the value at the previous frame, $v_c(t)$ the value received at instant t, Q the quantization process, IQ the inverse quantization, aa the arithemtic encoder operation, aa-1 the inverse operation. The value a time t is obtained from the previous value in the following way:

$$\hat{v} = V_{\min} + \frac{V_{\max} - V_{\min}}{2^{Nb} - 1} v_q$$

This formula applies for all animation types except normals (type 4) and rotation (type 6). For normals, the formula is applied only within each octant for computing the x and y coordinates on the unit square, so that the x and y values get quantized and encoded by the arithmetic encoder. Moreover, **only the Intra formula** is applied for these values. For rotations (type 6), the value is obtained by applying the encoding to the axis (if hasAxis =1) as a normal, and then to the angle (if hasAngle =1) separately.

For the arithmetic encoding and decoding, each field maintains its own statistics. I and P frames use the same statistics.

9.5 Node Semantics

9.5.1 Shared Nodes

9.5.1.1 Shared Nodes Overview

The Shared nodes are those nodes which may be used in both 2D and 3D scenes.

9.5.1.2 Shared Native Nodes

The following nodes are specific to this Final Committee Draft of International Standard.

9.5.1.2.1 AnimationStream

9.5.1.2.1.1 Semantic Table

```
AnimationStream {
    exposedField   SFBool     loop        FALSE
    exposedField   SFFloat    speed       1
    exposedField   SFTime     startTime   0
    exposedField   SFTime     stopTime    0
    exposedField   MFString   url         [""]
    eventOut       SFBool     isActive
}
```

9.5.1.2.1.2 Main Functionality

The **AnimationStream** node is designed to implement control parameters for a BIFS-Anim stream.

9.5.1.2.1.3 Detailed Semantic

The **loop**, **startTime**, and **stopTime exposedFields** and the **isActive eventOut**, and their effects on the **AnimationStream** node are described in Subclause 9.2.13.

The **speed** exposedField controls playback speed. An **AnimationStream** shall have no effect if **speed** is 0. For positive values of **speed**, the animation frame that an active **AnimationStream** will process at time *now* corresponds to the animation frame at BIFS-Anim time (i.e,. in the BIFS-Anim elementary stream's time base with animation frame 0 at time 0, at speed = 1):

fmod (now - startTime, duration/speed)

If**speed** is negative, then the animation frame to process is that at BIFS-Anim time:

duration + fmod(now - startTime, duration/speed).

For streaming media, negative values of the **speed** field are not implementable and shall be ignored.

When an **AnimationStream** becomes inactive, no animation frames are processed. The **speed** exposedField indicates how fast the BIFS-Anim stream should be processed. A speed of 2 indicates the stream plays twice as fast. Note that the **duration_changed** eventOut is not affected by the **speed** exposedField. **set_speed events** shall be ignored while the stream is playing.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1).

9.5.1.2.2 AudioDelay

The AudioDelay node allows sounds to be started and stopped under temporal control. The start time and stop time of the child sounds are delayed or advanced accordingly.

9.5.1.2.2.1 Semantic Table

```
AudioDelay {
    exposedField   MFNode     children     NULL
    exposedField   SFTime     delay        0
    field          SFInt32    numChan      1
    field          MFInt32    phaseGroup   NULL
}
```

9.5.1.2.2.2 Main Functionality

This node is used to delay a group of sounds, so that they start and stop playing later than specified in the AudioSource nodes.

9.5.1.2.2.3 Detailed Semantics

The **children** array specifies the nodes affected by the delay.

The **delay** field specifies the delay to apply to each chld.

The **numChan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.13.

9.5.1.2.2.4 Calculation

Implementation of the **AudioDelay** node requires the use of a buffer of size $d * S * n$, where d is the length of the delay in seconds, S is the sampling rate of the node, and n is the number of output channels from this node. At scene startup, a multichannel delay line of length d and width n is initialized to reside in this buffer

At each time step, the $k * S$ audio samples in each channel of the input buffer, where k is the length of the system time step in seconds, are inserted into this delay line. If the number of input channels is strictly greater than the number of output channels, the extra input channels are ignored; if the number of input channels is strictly less than the number of output channels, the extra channels of the delay line shall be taken as all 0's.

The output buffer of the node is the $k * S$ audio samples which fall off the end of the delay line in this process. Note that this definition holds regardless of the relationship between k and d.

If the **delay** field is updated during playback, discontinuties (audible artefacts or "clicks") in the output sound may result. If the **delay** field is updated to a greater value than the current value, the delay line is immediately extended to the new length, and zero values inserted at the beginning, so that $d * S$ seconds later there will be a short gap in the output of the node. If the **delay** field is updated to a lesser value than the current value, the delay line is immediately shortened to the new length, truncating the values at the end of of the line, so that there is an immediate discontinuity in sound output. Manipulation of the **delay** field in this manner is not recommended unless the audio is muted within the decoder or by appropriate use of an **AudioMix** node at the same time, since it gives rise to impaired sound quality.

9.5.1.2.3 AudioMix

9.5.1.2.3.1 Semantic Table

```
AudioMix {
    exposedField   MFNode    children      NULL
    exposedField   SFInt32   numInputs     1
    exposedField   MFFloat   matrix        NULL
    field          SFInt32   numChan       1
    field          MFInt32   phaseGroup    NULL
}
```

9.5.1.2.3.2 Main Functionality

This node is used to mix together several audio signals in a simple, multiplicative way. Any relationship that may be specified in terms of a mixing matrix may be described using this node.

9.5.1.2.3.3 Detailed Semantics

The **children** field specifies which nodes' outputs to mix together.

The **numInputs** field specifies the number of input channels. It should be the sum of the number of channels of the children.

The **matrix** array specifies the mixing matrix which relates the inputs to the outputs. **matrix** is an unrolled **numInputs** x **numChan** matrix which describes the relationship between **numInputs** input channels and **numChan** output channels. The **numInputs** * **numChan** values are in row-major order. That is, the first **numInputs** values are the scaling factors applied to each of the inputs to produce the first output channel; the next **numInputs** values produce the second output channel, and so forth.

That is, if the desired mixing matrix is $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$, specifying a "2 into 3" mix, the value of the **matrix** field should be [*a b c d e f*]**.**

The **numchan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.13.

9.5.1.2.3.4 Calculation

The value of the output buffer for an **AudioMix** node is calculated as follows. For each sample number *x* of output channel *i*, 1 <= *i* <= **numChan**, the value of that sample is

matrix[(0) * **numChan** + **i**] * **input**[1][*x*] +

matrix[(1) * **numChan** + **i**] * **input**[2][*x*] + ...

matrix[(**numInputs** – 1) * **numChan** + **i**] * **input**[**numInputs**][*x*],

where **input**[*i*][*j*] represents the *j*th sample of the *i*th channel of the input buffer, and the **matrix** elements are indexed starting from 1.

9.5.1.2.4 AudioSource

9.5.1.2.4.1 Semantic Table

AudioSource {

exposedField	MFNode	**children**	NULL
exposedField	MFString	**url**	NULL
exposedField	SFFloat	**pitch**	1
exposedField	SFFloat	**speed**	1
exposedField	SFTime	**startTime**	0
exposedField	SFTime	**stopTime**	0
field	SFInt32	**numChan**	1
field	MFInt32	**phaseGroup**	NULL

}

9.5.1.2.4.2 Main Functionality

This node is used to add sound to a BIFS scene. See Part 3 of this Final Committee Draft of International Standard for information on the various audio tools available for coding sound.

9.5.1.2.4.3 Detailed Semantics

The **children** field allows buffered **AudioClip** data to be used as sound samples within a Structured Audio decoding process. Only **AudioClip** nodes shall be children to an **AudioSource** node, and only in the case where **url** indicates a Structured Audio bitstream.

The **pitch** field controls the playback pitch for the Parametric and Structured Audio decoders. It is specified as a ratio, where 1 indicates the original bitstream pitch, values other than 1 indicate pitch-shifting by the given ratio. This field controls the Parametric decoder directly (see Clause 4 of Part 3 of this Final Committee Draft of International Standard); it is available as the globalPitch variable in the Structured Audio decoder (see Subclause 5.5.5 of Part 3 of this Final Committee Draft of International Standard).

The **speed** field controls the playback speed for the Parametric and Structured Audio decoders (see Part 3 of this Final Committee Draft of International Standard). It is specified as a ratio, where 1 indicates the original speed; values other than 1 indicate multiplicative time-scaling by the given ratio (i.e. 0.5 specifies twice as fast). The value of this field is always made available to the Structured Audio or Parametric audio decoder indicated by the **url** field. Subclauses XXX and XXX, in Part 3 of this Final Committee Draft of International Standard, describe the use of this field to control the Parametric and Structured Audio decoders respectively.

Kommentar: To be provided by Eric in the next iteration.

The **startTime** field specifies a time at which to start the audio playing.

The **stopTime** field specifies a time at which to turn off the Sound. Sounds which have limited extent in time turn themselves off when the decoder finished. If the **stopTime** field is 0, the Sound continues until it is finished or plays forever.

The **numChan** field describes how many channels of audio are in the decoded bitstream.

The **phaseGroup** array specifies whether or not there are important phase relationships between the multiple channels of audio. If there are such relationships – for example, if the sound is a multichannel spatialized set or a "stereo pair" – it is in general dangerous to do anything more complex than scaling to the sound. Further filtering or repeated "spatialization" will destroy these relationships. The values in the array divide the channels of audio into groups; if **phaseGroup[i] = phaseGroup[j]** then channel **i** and channel **j** are phase-related. Channels for which the **phaseGroup** value is 0 are not related to any other channel.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1).

9.5.1.2.4.4 Calculation

The audio output from the decoder according to the bitstream(s), referenced in the specified URL, at the current scene time is placed in the output buffer for this node, unless the current scene time is earlier than the current value of **startTime** or later than the current value of **stopTime**, in which case 0 values are placed in the output buffer for this node for the current scene time.

For audio sources decoded using the Structured Audio decoder (Clause 5 in Part 3 of this Final Committee Draft of International Standard) Profile 4, several variables from the scene description must be mapped into standard names in the orchestra. See Subclause 5.11 in Part 3 of this Final Committee Draft of International Standard.

If **AudioClip** children are provided for a structured audio decoder, the audio data buffered in the **AudioClip**(s) must be made available to the decoding process. See Subclause 5.6.2 in Part 3 of this Final Committee Draft of International Standard.

9.5.1.2.5 AudioFX

9.5.1.2.5.1 Semantic Table

AudioFX {

exposedField	MFNode	**children**	NULL
exposedField	SFString	**orch**	""
exposedField	SFString	**score**	""
exposedField	MFFloat	**params**	NULL
field	SFInt32	**numChan**	1
field	MFInt32	**phaseGroup**	NULL

}

9.5.1.2.5.2 Main Functionality

The **AudioFX** node is used to allow arbitrary signal-processing functions defined using Structured Audio tools to be included and applied to its **children** (see Part 3 of this Final Committee Draft of International Standard).

9.5.1.2.5.3 Detailed Semantics

The **children** array contains the nodes operated upon by this effect. If this array is empty, the node has no function (the node may not be used to create new synthetic audio in the middle of a scene graph).

The **orch** string contains a tokenised block of signal-processing code written in SAOL (Structured Audio Orchestra Language). This code block should contain an orchestra header and some instrument definitions, and conform to the bitstream syntax of the orchestra class as defined in Part 3 of this Final Committee Draft of International Standard, Subclauses 5.1 and 5.4.

Kommentar: From Eric: There is a bug here because the tokenized code block may contain zeros, but this field is zero-delimited right now.

The **score** string may contain a tokenized score for the given orchestra written in SASL (Structured Audio Score Language). This score may contain control operators to adjust the parameters of the orchestra, or even new instrument instantiations. A score is not required; if present it shall conform to the bitstream syntax of the score_file class as defined in Subclause 5.1 of Part 3 of this Final Committee Draft of International Standard.

The **params** field allows BIFS updates and events to affect the sound-generation process in the orchestra. The values of **params** are available to the FX orchestra as the global array “global ksig params[128]”; see Subclause 5.11 of Part 3 of this Final Committee Draft of International Standard.

The **numchan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see Subclause 9.2.13.

9.5.1.2.5.4 Calculation

The node is evaluated according to the semantics of the orchestra code contained in the **orch** field. See Clause 5 of Part 3 of this Final Committee Draft of International Standard for the normative parts of this process, especially Subclause 5.11. Within the orchestra code, the multiple channels of input sound are placed on the global bus, **input_bus**; first, all channels of the first child, then all the channels of the second child, and so on. The orchestra header should ‘send’ this bus to an instrument for processing. The **phaseGroup** arrays of the children are made available as the inGroup variable within the instrument(s) to which the **input_bus**is sent.

The orchestra code block shall not contain the **spatialize** statement.

The output buffer of this node is the sound produced as the final output of the orchestra applied to the input sounds, as described in Part 3 of this Final Committee Draft of International Standard, Subclause 5.3.3.3.6.

Kommentar: Check...

9.5.1.2.6 AudioSwitch

9.5.1.2.6.1 Semantic Table

AudioSwitch {

exposedField	MFNode	**children**	NULL
exposedField	MFInt32	**whichChoice**	NULL
field	SFInt32	**numChan**	1
field	MFInt32	**phaseGroup**	NULL

}

9.5.1.2.6.2 Main Functionality

The **AudioSwitch** node is used to select a subset of audio channels from the child nodes specified.

9.5.1.2.6.3 Detailed Semantics

The **children** field specifies a list of child options.

The **whichChoice** field specifies which channels should be passed through. If **whichChoice[i]** is 1, then the i-th child channel should be passed through.

The **numchan** field specifies the number of channels of audio output by this node; ie, the number of channels in the passed child.

The **phaseGroup** field specifies the phase relationships among the various output channels; see Subclause 9.2.13.

9.5.1.2.6.4 Calculation

The values for the output buffer are calculated as follows:

For each sample number x of channel number i of the output buffer, $1 <= i <=$ **numChan**, the value in the buffer is the same as the value of sample number x in the jth channel of the input, where j is the least value such that **whichChoice[0] + whichChoice[1] + ... + whichChoice[*j*] =** i.

9.5.1.2.7 Conditional

9.5.1.2.7.1 Semantic Table

Conditional {

eventIn	SFBool	**activate**	FALSE
eventIn	SFBool	**reverseActivate**	FALSE
exposedField	SFString	**buffer**	""
eventOut	SFBool	**isActive**	

}

9.5.1.2.7.2 Main Functionality

A **Conditional** node interprets a buffered bit stream when it is activated. This allows events to trigger node updates, deletions, and other modifications to the scene. The buffered bit stream is interpreted as if it had just been received. The typical use of this node for the implementation of the action of a button is the following: the button geometry is enclosed in a grouping node which also contains a **TouchSensor** node. The **isActive** eventOut of the **TouchSensor** is routed to the activate eventIn of **Conditional** C1 and to the **reverseActivate** eventIn of **Conditional** C2; C1 then implements the "mouse-down" action and C2 implements the "mouse-up" action.

9.5.1.2.7.3 Detailed Semantics

Upon reception of either an SFBool event of value TRUE on the **activate** eventIn, or an SFBool event of value FALSE on the **reverseActivate** eventIn, the contents of the buffer field are interpreted as BIFS updates. These updates are not time-stamped; they are executed at the time of the event.

9.5.1.2.8 MediaTimeSensor

9.5.1.2.8.1 Semantic Table

MediaTimeSensor {

exposedField	SFNode	**media**	NULL
field	SFNode	**timer**	NULL

}

9.5.1.2.8.2 Main Functionality

The **MediaTimeSensor** node provides a mechanism to attach a media stream to a **TimeSensor** node, slaving the timer to the media stream's timebase.

9.5.1.2.8.3 Detailed Semantics

The **MediaTimeSensor** is a way to link a **TimeSensor** clock to a specific streaming media clock. The **media** field is a pointer to a node that is linked to a streaming media. All the SFTime values in the attached **TimeSensor** node will then be interpreted as values related to the conveyed time base of the pointed stream. This enables in particular to start an animation after a given time that a media stream is streaming, whether it has been stopped or not. If the value of media is NULL, then the time events in the **TimeSensor**will be refering to the time base used by the BIFS stream.

9.5.1.2.9 QuantizationParameter

9.5.1.2.9.1 Semantic Table

QuantizationParameter {

field	SFBool	**isLocal**	FALSE
field	SFBool	**position3DQuant**	FALSE
field	SFVec3f	**position3DMin**	$-\infty, -\infty, -\infty$
field	SFVec3f	**position3DMax**	$+\infty, +\infty, +\infty$
field	SFInt32	**position3DNbBits**	16
field	SFBool	**position2DQuant**	FALSE
field	SFVec2f	**position2DMin**	$-\infty, -\infty$
field	SFVec2f	**position2DMax**	$+\infty, +\infty$
field	SFInt32	**position2DNbBits**	16
field	SFBool	**drawOrderQuant**	TRUE

field	SFVec3f	**drawOrderMin**	-∞
field	SFVec3f	**drawOrderMax**	+∞
field	SFInt32	**drawOrderNbBits**	8
field	SFBool	**colorQuant**	TRUE
field	SFFloat	**colorMin**	0
field	SFFloat	**colorMax**	1
field	SFInt32	**colorNbBits**	8
field	SFBool	**textureCoordinateQuant**	TRUE
field	SFFloat	**textureCoordinateMin**	0
field	SFFloat	**textureCoordinateMax**	1
field	SFInt32	**textureCoordinateNbBits**	16
field	SFBool	**angleQuant**	TRUE
field	SFFloat	**angleMin**	0
field	SFFloat	**angleMax**	2π
field	SFInt32	**angleNbBits**	16
field	SFBool	**scaleQuant**	TRUE
field	SFFloat	**scaleMin**	0
field	SFFloat	**scaleMax**	+∞
field	SFInt32	**scaleNbBits**	8
field	SFBool	**keyQuant**	TRUE
field	SFFloat	**keyMin**	0
field	SFFloat	**keyMax**	1
field	SFInt32	**keyNbBits**	8
field	SFBool	**normalQuant**	TRUE
field	SFInt32	**normalNbBits**	8

}

9.5.1.2.9.2 Main Functionality

The **QuantizationParameter** node describes the quantization values to be applied on single fields of numerical types. For each of identified categories of fields, a minimal and maximal value is given as well as a number of bits to represent the given class of fields. Additionally, it is possible to set the **isLocal** field to apply the quantization only to the node following the **QuantizationParameter** node. The use of a node structure for declaring the quantization parameters allows the application of the DEF and USE mechanisms that enable reuse of the **QuantizationParameter** node. Also,it enables the parsing of this node in the same manner as any other scene information.

9.5.1.2.9.3 Detailed Semantics

The **QuantizationParameter** node can only appear as a child of a grouping node. When a **QuantizationParameter** node appears in the scene graph, the quantization is set to TRUE, and will apply to the following nodes as follows:

- If the **isLocal** boolean is set to FALSE, the quantization applies to all siblings following the **QaunitzationParameter** node, and thus to all their children as well.
- If the **isLocal** boolean is set to TRUE, the quantization only applies to the following sibling node in the children list of the parent node. If no sibling is following the **QuantizationParameter** node declaration, the node has no effect.
- In all cases, the quantization is applied only in the scope of a single BIFS command. That is, if a command in the same access unit, or in another access unit inserts a node in a context in which the quantization was active, no quantization will be applied, except if a new **QuantizationParameter** node is defined in this new command.

The information contained in the **QuantizationParameter** node fields applies within the context of the node scope as follows. For each category of fields, a boolean sets the quantization on and off, the minimal and maximal values are set, as well as the number of bits for the quantization. This information, combined with the node coding table, enables the relevant information to quantize the fields to be obtained. The quantization parameters are applied as explained in Subclause 9.4.1.10.

9.5.1.2.10 TermCap

9.5.1.2.10.1 Semantic Table

TermCap {

eventIn	SFTime	**evaluate**	NULL
field	SFInt	**capability**	0
eventOut	SFInt	**value**	

}

9.5.1.2.10.1.1 Main functionality

The **TermCap** node is used to query the resources of the terminal. By ROUTEing the result to a **Switch** node, simple adaptive content may be authored using BIFS.

9.5.1.2.10.1.2 Detailed Semantics

When this node is instantiated, the value of the **capability** field shall be examined by the system and the **value** eventOut generated to indicate the associated system capability. The **value** eventOut is updated and generated whenever an **evaluate** eventIn is received.

The **capability** field specifies a terminal resource to query. The semantics of the **value** field vary depending on the value of this field. The capabilities which may be queried are:

Value	Interpretation
0	frame rate
1	color depth
2	screen size
3	graphics hardware
32	audio output format
33	maximum audio sampling rate
34	spatial audio capability
64	CPU load
65	memory load

The exact semantics differ depending on the value of the **capability** field, as follows

capability: 0 (frame rate)

For this value of **capability**, the current rendering frame rate is measured. The exact method of measurement not specified.

Value	Interpretation
0	unknown or can't determine
1	less than 5 fps
2	5-10 fps
3	10-20 fps
4	20-40 fps
5	more than 40 fps

For the breakpoint between overlapping values between each range (i.e. 5, 10, 20, and 40), the higher value of **value** shall be used (ie, 2, 3, 4, and 5 respectively). This applies to each of the subsequent **capability-value** tables as well.

capability: 1 (color depth)

For this value of **capability**, the color depth of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the color depth as follows:

Value	Interpretation
0	unknown or can't determine
1	1 bit/pixel

2	grayscale
3	color, 3-12 bit/pixel
4	color, 12-24 bit/pixel
5	color, more than 24 bit/pixel

capability: 2 (screen size)

For this value of capability, the window size (in horizontal lines) of the output window of the rendering terminal is measured:

Value	Interpretation
0	unknown or can't determine
1	less than 200 lines
2	200-400 lines
3	400-800 lines
4	800-1600 lines
5	1600 or more lines

capability: 3 (graphics hardware)

For this value of **capability**, the available of graphics acceleration hardware of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the available graphics hardware:

Value	Interpretation
0	unknown or can't determine
1	no acceleration
2	matrix multiplication
3	matrix multiplication + texture mapping (less than 1M memory)
4	matrix multiplication + texture mapping (less than 4M memory)
5	matrix multiplication + texture mapping (more than 4M memory)

capability: 32 (audio output format)

For this value of **capability**, the audio output format (speaker configuration) of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the audio output format.

Value	Interpretation
0	unknown or can't determine
1	mono
2	stereo speakers
3	stereo headphones
4	five-channel surround
5	more than five speakers

capability: 33 (maximum audio sampling rate)

For this value of **capability**, the maximum audio output sampling rate of the rendering terminal is measured. At the time this node is instantiated, the value field is set to indicate the maximum audio output sampling rate.

Value	Interpretation
0	unknown or can't determine
1	less than 16000 Hz
2	16000-32000 Hz

3	32000-44100 Hz
4	44100-48000 Hz
5	48000 Hz or more

capability: 34 (spatial audio capability)

For this value of **capability**, the spatial audio capability of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the spatial audio capability.

Value	Interpretation
0	unknown or can't determine
1	no spatial audio
2	panning only
3	azimuth only
4	full 3-D spatial audio

capability: 64 (CPU load)

For this value of **capability**, the CPU load of the rendering terminal is measured. The exact method of measurement is not specified. The value of the **value** eventOut indicates the available CPU resources as a percentage of the maximum available; that is, if all of the CPU cycles are being consumed, and no extra calculation can be performed without compromising real-time performance, the indicated value is 100%; if twice as much calculation as currently being done can be so performed, the indicated value is 50%.

Value	Interpretation
0	unknown or can't determine
1	less than 20% loaded
2	20-40% loaded
3	40-60% loaded
4	60-80% loaded
5	80-100% loaded

capability: 65 (RAM available)

For this value of **capability**, the available memory of the rendering terminal is measured. The exact method of measurement is not specified.

Value	Interpretation
0	unknown or can't determine
1	less than 100 KB free
2	100 KB – 500 KB free
3	500 KB – 2 MB free
4	2 MB – 8 MB free
5	8 MB – 32 MB free
6	32 MB – 200 MB free
7	more than 200 MB free

9.5.1.2.11 Valuator

9.5.1.2.11.1 Semantic Table

Valuator {

eventIn	SFBool	**inSFBool**	NULL
eventIn	SFColor	**inSFColor**	NULL
eventIn	MFColor	**inMFColor**	NULL
eventIn	SFFloat	**inSFFloat**	NULL
eventIn	MFFloat	**inMFFloat**	NULL
eventIn	SFInt32	**inSFInt32**	NULL
eventIn	MFInt32	**inMFInt32**	NULL
eventIn	SFRotation	**inSFRotation**	NULL
eventIn	MFRotation	**inMFRotation**	NULL

eventIn	SFString	**inSFString**	NULL
eventIn	MFString	**inMFString**	NULL
eventIn	SFTime	**inSFTime**	NULL
eventIn	SFVec2f	**inSFVec2f**	NULL
eventIn	MFVec2f	**inMFVec2f**	NULL
eventIn	SFVec3f	**inSFVec3f**	NULL
eventIn	MFVec3f	**inMFVec3f**	NULL
exposedField	SFBool	**outSFBool**	FALSE
exposedField	SFColor	**outSFColor**	0, 0, 0
exposedField	MFColor	**outMFColor**	NULL
exposedField	SFFloat	**outSFFloat**	0
exposedField	MFFloat	**outMFFloat**	NULL
exposedField	SFInt32	**outSFInt32**	0
exposedField	MFInt32	**outMFInt32**	NULL
exposedField	SFRotation	**outSFRotation**	0
exposedField	MFRotation	**outMFRotation**	NULL
exposedField	SFString	**outSFString**	""
exposedField	MFString	**outMFString**	NULL
exposedField	SFTime	**outSFTime**	0
exposedField	SFVec2f	**outSFVec2f**	0, 0
exposedField	MFVec2f	**outMFVec2f**	NULL
exposedField	SFVec3f	**outSFVec3f**	0, 0, 0
exposedField	MFVec3f	**outMFVec3f**	NULL

}

9.5.1.2.11.2 Main Functionality

A Valuator node can receive an event of any type, and on reception of such an event, will trigger eventOuts of many types with given values. It can be seen as an event type adapter. One use of this node is the modification of the SFInt **whichChoice** field of a **Switch** node by an event. There is no interpolator or sensor node with an **SFInt eventOut**. Thus, if a two-state button is described with a **Switch** containing the description of each state in choices 0 and 1, the triggering event of any type can be routed to a Valuator node which **outInt** field is set to 1 and routed to the **whichChoice** field of the Switch. The return to the 0 state needs another **Valuator** node.

9.5.1.2.11.3 Detailed Semantics

Upon reception of an event on any of its eventIns, on each eventOut connected to a ROUTE an event will be generated. The value of this event is the current value of the eventOut. Note that the value of the eventOuts bears no relationship with the value of the eventIn, even if their types are the same. As such, this node does not do type casting.

9.5.1.3 Shared VRML Nodes

The following nodes have their semantic specified in ISO/IEC 14772-1 with further restrictions and extensions defined herein.

9.5.1.3.1 Appearance

9.5.1.3.1.1 Semantic Table

Appearance {

exposedField	SFNode	**material**	NULL
exposedField	SFNode	**texture**	NULL
exposedField	SFNode	**textureTransform**	NULL

}

9.5.1.3.2 AudioClip

9.5.1.3.2.1 Semantic Table

AudioClip {

exposedField	SFBool	**loop**	FALSE
exposedField	SFFloat	**pitch**	1
exposedField	SFTime	**startTime**	0
exposedField	SFTime	**stopTime**	0
exposedField	MFString	**url**	NULL
eventOut	SFTime	**duration_changed**	
eventOut	SFBool	**isActive**	

}

9.5.1.3.2.2 Main Functionality

The **AudioClip** node is used to provide an interface for short snippets of audio to be used in an interactive scene, such as sounds triggered as “auditory icons” upon mouse clicks. It buffers up the audio generated by its children, so that it can provide random restart capability upon interaction.

The **AudioClip** node provides a special “buffering” interface to streaming audio, to convert it into a non-streaming form so that it can be used interactively, such as for auditory feedback of event triggers or other interactive “sound effect” processes.

9.5.1.3.2.3 Calculation

The output of this node is not calculated based on the current input values, but according to the **startTime** event, the **pitch** field and the contents of the clip buffer. When the **startTime** is reached, the sound output begins at the beginning of the clip buffer and **isActive** is set to 1. At each time step thereafter, the value of the output buffer is the value of the next portion of the clip buffer, upsampled or downsampled as necessary according to **pitch**. When the end of the clip buffer is reached, if **loop** is set, the audio begins again from the beginning of the clip buffer; if not, the playback ends.

The clip buffer is calculated as follows: when the node is instantiated, for the first *length [units]*, the audio input to this node is copied into the clip buffer; that is, after t seconds, where $t < length$, audio sample number $t * S$ of channel i in the buffer contains the audio sample corresponding to time t of channel i of the input, where S is the sampling rate of this node. After the first *length [units]*, the input to this node has no effect.

When the playback ends, either because **stopTime** is reached or because the end of the clip buffer is reached for a non-looping clip, the **isActive** field is set to 0.

When the playback is not active, the audio output of the node is all 0s.

If**pitch** is negative, the buffer is played backwards, beginning with the last segment.

9.5.1.3.3 Color

9.5.1.3.3.1 Semantic Table

Color {

exposedField	MFColor	**color**	NULL

}

9.5.1.3.4 ColorInterpolator

9.5.1.3.4.1 Semantic Table

ColorInterpolator {

eventIn	SFFloat	**set_fraction**	NULL
exposedField	MFFloat	**key**	NULL
exposedField	MFColor	**keyValue**	NULL

```
    eventOut      SFColor     value_changed
}
```

9.5.1.3.5 FontStyle

9.5.1.3.5.1 Semantic Table

```
FontStyle {
    field         MFString    family          ["SERIF"]
    field         SFBool      horizontal      TRUE
    field         MFString    justify         ["BEGIN"]
    field         SFString    language        ""
    field         SFBool      leftToRight     TRUE
    field         SFFloat     size            1
    field         SFFloat     spacing         1
    field         SFString    style           "PLAIN"
    field         SFBool      topToBottom     TRUE
}
```

9.5.1.3.6 ImageTexture

9.5.1.3.6.1 Semantic Table

```
ImageTexture {
    exposedField  MFString    url             NULL
    field         SFBool      repeatS         TRUE
    field         SFBool      repeatT         TRUE
}
```

9.5.1.3.6.2 Detailed Semantics

The **url** field specifies the data source to be used (see 9.2.7.1).

9.5.1.3.7 MovieTexture

9.5.1.3.7.1 Semantic Table

```
MovieTexture {
    exposedField  SFBool      loop              FALSE
    exposedField  SFFloat     speed             1
    exposedField  SFTime      startTime         0
    exposedField  SFTime      stopTime          0
    exposedField  MFString    url               NULL
    field         SFBool      repeatS           TRUE
    field         SFBool      repeatT           TRUE
    eventOut      SFTime      duration_changed
    eventOut      SFBool      isActive
}
```

9.5.1.3.7.2 Detailed Semantics

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **MovieTexture** node, are described in Subclause 9.2.13. The **speed** exposedField controls playback speed. A **MovieTexture** shall display frame or VOP 0 if speed is 0. For positive values of **speed**, the frame or VOP that an active **MovieTexture** will display at time *now* corresponds to the frame or VOP at movie time (i.e., in the movie's local time base with frame or VOP 0 at time 0, at speed = 1):

fmod (now - startTime, duration/speed)

If**speed** is negative, then the frame or VOP to display is the frame or VOP at movie time:

duration + fmod(now - startTime, duration/speed).

A **MovieTexture** node is inactive before **startTime** is reached. If **speed** is non-negative, then the first VOP shall be used as texture, if it is already available. If **speed** is negative, then the last VOP shall be used as texture, if it is already available. For streaming media, negative values of **speed** are not implementable and shall be ignored.

When a **MovieTexture** becomes inactive, the VOP corresponding to the time at which the **MovieTexture** became inactive shall persist as the texture. The **speed** exposedField indicates how fast the movie should be played. A speed of 2 indicates the movie plays twice as fast. Note that the **duration_changed** eventOut is not affected by the **speed** exposedField. **set_speed** events shall be ignored while the movie is playing.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1).

9.5.1.3.8 ScalarInterpolator

9.5.1.3.8.1 Semantic Table

```
ScalarInterpolator {
    eventIn         SFFloat     set_fraction        NULL
    exposedField    MFFloat     key                 NULL
    exposedField    MFFloat     keyValue            NULL
    eventOut        SFFloat     value_changed
}
```

9.5.1.3.9 Shape

9.5.1.3.9.1 Semantic Table

```
Shape {
    exposedField    SFNode      appearance          NULL
    exposedField    SFNode      geometry            NULL
}
```

9.5.1.3.10 Switch

9.5.1.3.10.1 Semantic Table

```
Switch {
    exposedField    MFNode      choice              NULL
    exposedField    SFInt32     whichChoice         -1
}
```

9.5.1.3.11 Text

9.5.1.3.11.1 Semantic Table

```
Text {
    exposedField    SFString    string              ""
    exposedField    MFFloat     length              NULL
    exposedField    SFNode      fontStyle           NULL
    exposedField    SFFloat     maxExtent           0
}
```

9.5.1.3.12 TextureCoordinate

9.5.1.3.12.1 Semantic Table

```
TextureCoordinate {
    exposedField   MFVec2f    point              NULL
}
```

9.5.1.3.13 TextureTransform

9.5.1.3.13.1 Semantic Table

```
TextureTransform {
    exposedField   SFVec2f    center             0, 0
    exposedField   SFFloat    rotation           0
    exposedField   SFVec2f    scale              1, 1
    exposedField   SFVec2f    translation        0, 0
}
```

9.5.1.3.14 TimeSensor

9.5.1.3.14.1 Semantic Table

```
TimeSensor {
    exposedField   SFTime     cycleInterval      1
    exposedField   SFBool     enabled            TRUE
    exposedField   SFBool     loop               FALSE
    exposedField   SFTime     startTime          0
    exposedField   SFTime     stopTime           0
    eventOut       SFTime     cycleTime
    eventOut       SFFloat    fraction_changed
    eventOut       SFBool     isActive
    eventOut       SFTime     time
}
```

9.5.1.3.15 TouchSensor

9.5.1.3.15.1 Semantic Table

```
TouchSensor {
    exposedField   SFBool     enabled            TRUE
    eventOut       SFVec3f    hitNormal_changed
    eventOut       SFVec3f    hitPoint_changed
    eventOut       SFVec2f    hitTexCoord_changed
    eventOut       SFBool     isActive
    eventOut       SFBool     isOver
    eventOut       SFTime     touchTime
}
```

9.5.1.3.16 WorldInfo

9.5.1.3.16.1 Semantic Table

```
WorldInfo {
    field          MFString   info               NULL
    field          SFString   title              ""
}
```

9.5.2 2D Nodes

9.5.2.1 2D Nodes Overview

The 2D nodes are those nodes which may be used in 2D scenes and with nodes that permit the use of 2D nodes in 3D scenes.

9.5.2.2 2D Native Nodes

9.5.2.2.1 Background2D

9.5.2.2.1.1 Semantic Table

Background2D {			
eventIn	SFBool	**set_bind**	NULL
exposedField	MFString	**url**	NULL
eventOut	SFBool	**isBound**	
}			

9.5.2.2.1.2 Main Functionality

There exists a **Background2D** stack, in which the top-most background is the current active background one. The **Background2D** node allows a background to be displayed behind a 2D scene. The functionality of this node can also be accomplished using other nodes, but use of this node may be more efficient in some implementations.

9.5.2.2.1.3 Detailed Semantics

If**set_bind** is set to TRUE the **Background2D** is moved to the top of the stack.

If **set_bind** is set to FALSE, the **Background2D** is removed from the stack so the previous background which is contained in the stack is on top again.

The **url** specifies the stream used for the backdrop.

The **isBound** event is sent as soon as the backdrop is put at the top of the stack, so becoming the current backdrop.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1).

This is not a geometry node and the top-left corner of the image is displayed at the top-left corner of the screen, regardless of the current transformation. Scaling and/or rotation do not have any effect on this node.

Example: Changing the background for 5 seconds.

```
Group2D {
   children [
      …
      DEF TIS TimeSensor {
         startTime 5.0
         stopTime  10.0
      }
      DEF BG1 Background2D {
         …
      }
   ]
}
ROUTE TIS.isActive TO BG1.set_bind
```

9.5.2.2.2 Circle

9.5.2.2.2.1 Semantic Table

Circle {			
exposedField	SFFloat	**radius**	1

}

9.5.2.2.2.2 Main Functionality

This node draws a circle.

9.5.2.2.2.3 Detailed Semantics

The **radius** field determines the radius of the rendered circle.

9.5.2.2.3 Coordinate2D

9.5.2.2.3.1 Semantic Table

```
Coordinate2D {
    exposedField   MFVec2f   point      NULL
}
```

9.5.2.2.3.2 Main Functionality

This node defines a set of 2D coordinates to be used in the **coord** field of geometry nodes.

9.5.2.2.3.3 Detailed Semantics

The **point** field contains a list of points in the 2D coordinate space. See Subclause 9.2.3.

9.5.2.2.4 Curve2D

9.5.2.2.4.1 Semantic Table

```
Curve2D {
    exposedField   SFNode    points     NULL
    exposedField   SFInt32   fineness   0
}
```

9.5.2.2.4.2 Main Functionality

This node is used to include the Bezier approximation of a polygon in the scene at an arbitrary level of precision. It behaves as other “lines”, which means it is sensitive to modifications of line width and “dotted-ness”, and can be filled or not.

The given parameters are a control polygon and a parameter setting the quality of approximation of the curve. Internally, another polygon of fineness points is computed on the basis of the control polygon. The coordinates of that internal polygon are given by the following formula:

$$x[j]=\sum_{i=0}^{n} xc[i]\times\frac{(n-1)!}{i!(n-1-i)!}\times\left(\frac{j}{f}\right)^{i}\times\left(1-\frac{j}{f}\right)^{n-1-i},$$

where *x[j]* is the jth x coordinate of the internal polygon, *n* is the number of points in the control polygon, *xc[i]* is the ith x coordinate of the control polygon and *f* is short for the above fineness parameter which is also the number of points in the internal polygon. A symmetrical formula yields the y coordinates.

9.5.2.2.4.3 Detailed Semantics

The **points** field lists the vertices of the control polygon. The **fineness** field contains the number of points in the internal polygon which constitutes the Bezier interpolation of the control polygon. **fineness** should be greater than the number of points in the control polygon.

Example: The following defines a 20-points Bezier approximation of a 4-point polygon.

```
geometry Curve2D {
   points Coordinate2D {
      point [ -10.00 0.00 0.00 50.00 15.00 25.00 25.00 15.00 ]
      fineness 20
   }
}
```

9.5.2.2.5 DiscSensor

9.5.2.2.5.1 Semantic Table

DiscSensor {

exposedField	SFBool	**autoOffset**	TRUE
exposedField	SFVec2f	**center**	0, 0
exposedField	SFBool	**enabled**	TRUE
exposedField	SFFloat	**maxAngle**	-1
exposedField	SFFloat	**minAngle**	-1
exposedField	SFFloat	**offset**	0
eventOut	SFBool	**isActive**	
eventOut	SFRotation	**rotation_changed**	
eventOut	SFVec3f	**trackPoint_changed**	

}

9.5.2.2.5.2 Main Functionality

This sensor enables the rotation of an object in the 2D plane around an axis specified in the local coordinate system.

9.5.2.2.5.3 Detailed Semantics

The semantics are as specified in ISO/IEC 14772-1, Subclause 6.15, but restricted to a 2D case.

9.5.2.2.6 Form

9.5.2.2.6.1 Semantic Table

Form {

field	MFNode	**children**	NULL
exposedField	SFVec2f	**size**	-1, -1
field	MFInt32	**groups**	NULL
field	MFInt32	**constraint**	NULL

}

9.5.2.2.6.2 Main Functionality

The **Form** node specifies the placement of its children according to relative alignment and distribution constraints. Distribution spreads objects regularly, with an equal spacing between them.

9.5.2.2.6.3 Detailed Semantics

The **children** field shall specify a list of nodes that are to be arranged. Note that the children's position is implicit and that order is important.

The **size** field specifies the width and height of the layout frame.

The **groups** field specifies the list of groups of objects on which the constraints can be applied. The children of the **Form** node are numbered from 1 to n, 0 being reserved for a reference to the layout itself. A group is a list of child indices, terminated by a -1.

The **constraints** field specifies the list of constraints. One constraint is constituted by a *constraint type*, optionally followed by a distance, followed by the indices of the objects and groups it is to be applied on and terminated by a –1. The numbering scheme is:

- 0 for a reference to the layout,
- 1 to n for a reference to one of the children,
- n+1 to n+m for a reference to one of the m specified groups.

Constraints belong to two categories: alignment and distribution constraints.

Components mentioned in the tables are components whose indices appear in the list following the constraint type. When rank is mentioned, it refers to the rank in that list. Spaces are specified in pixels, positive from left to right and bottom to top.

Table 9-4: Alignment Constraints

Alignment Constraints	Type Index	Effect
AL: Align Left edges	0	The xmin of constrained components become equal to the xmin of the left-most component.
AH: Align centers Horizontally	1	The (xmin+xmax)/2 of constrained components become equal to the (xmin+xmax)/2 of the group of constrained components as computed before this constraint is applied.
AR: Align Right edges	2	The xmax of constrained components become equal to the xmax of the right-most component.
AT: Align Top edges	3	The ymax of all constrained components become equal to the ymax of the top-most component.
AV: Align centers Vertically	4	The (ymin+ymax)/2 of constrained components become equal to the (ymin+ymax)/2 of the group of constrained components as computed before this constraint is applied.
AB: Align Bottom edges	5	The ymin of constrained components become equal to the ymin of the bottom-most component.
ALspace: Align Left edges by specified space	6	The xmin of the second and following components become equal to the xmin of the first component plus the specified space.
ARspace: Align Right edges by specified space	7	The xmax of the second and following components become equal to the xmax of the first component minus the specified space.
ATspace: Align Top edges by specified space	8	The ymax of the second and following components become equal to the ymax of the first component minus the specified space.
ABspace: Align Bottom edges by specified space	9	The ymin of the second and following components become equal to the ymin of the first component plus the specified space.

The purpose of distribution constraints is to specify the space between components, by making such pairwise gaps equal either to a given value or to the effect of filling available space.

Table 9-5: Distribution Constraints

Distribution Constraints	Type Index	Effect
SH: Spread Horizontally	10	The differences between the xmin of each component and the xmax of the previous one become all equal. The first and the last component should be constrained horizontally already.
SHin: Spread Horizontally in container	11	The differences between the xmin of each component and the xmax of the previous one become all equal. References are the edges of the layout.
SHspace: Spread Horizontally by specified space	12	The difference between the xmin of each component and the xmax of the previous one become all equal to the specified space. The first component is not moved.
SV: Spread Vertically	13	The differences between the ymin of each component and the ymax of the previous one become all equal. The first and the last

		component should be constrained vertically already.
SVin: Spread Vertically in container	14	The differences between the ymin of each component and the ymax of the previous one become all equal. References are the edges of the layout.
SVspace: Spread Vertically by specified space	15	The difference between the ymin of each component and the ymax of the previous one become all equal to the specified space. The first component is not moved.

All objects start at the center of the Form. The constraints are then applied in sequence.

Example: Laying out five 2D objects.

```
Shape {
   Geometry2D Rectangle { size 50 55 } // draw the Form's frame.
   VisualProps use VPSRect
}

Transform2D {
   translation 10 10 {
      children [
            Form {
            children [
               Shape2D { use OBJ1 }
               Shape2D { use OBJ2 }
               Shape2D { use OBJ3 }
               Shape2D { use OBJ4 }
               Shape2D { use OBJ5 }
            ]
            size 50 55
                        groups [ 1 3 -1]
                        constraints [11 6 -1 14 1 -1 2 0 2 -1 5 0 2 -1 9 6 0 3 -1
                                    9 7 0 4 -1 6 7 0 4 -1 8 -2 0 5 -1 7 -2 0 5 -1

                        ]
         }
      ]
   }
}
```

The above **constraints** specify the following operations:

- spread horizontally in container object 6 = the group 1,3
- spread vertically in container object 1
- align the right edge of objects 0 (container) and 2
- align the bottom edge of the container and object 2
- align the bottom edge with space 6 the container and object 3
- align the bottom edge with space 7 the container and object 4
- align the left edge with space 7 the container and object 4
- align the top edge with space -2 the container and object 5
- align the right edge with space -2 the container and object 5

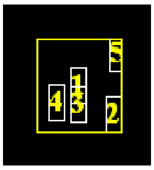


Figure 9-9: Visual result of the Form node example

9.5.2.2.7 Group2D

9.5.2.2.7.1 Semantic Table

Group2D {

eventIn	MFNode	**addChildren**	NULL
eventIn	MFNode	**removeChildren**	NULL
exposedField	MFNode	**children**	NULL
field	SFVec2f	**bboxCenter**	0, 0
field	SFVec2f	**bboxSize**	-1, -1

}

9.5.2.2.7.2 Main Functionality

The **Group2D** node is one of the grouping 2D nodes. It is, itself, a 2D scene. A **Group2D** node contains children nodes without introducing any transformation.

9.5.2.2.7.3 Detailed Semantics

The **addChildren** eventIn specifies a list of 2D objects that must be added to the Group2D node.

The **removeChildren** eventIn specifies a list of 2D objects that must be removed from the Group2D node.

The **children** field is the current list of 2D objects contained in the **Group2D** node.

The **bboxCenter** field specifies the center of the bounding box and the **bboxSize** field specifies the width and the height of the bounding box. It is possible not to transmit the **bboxCenter** and **bboxSize** fields, but if they are transmitted, the corresponding box must contain all the children. The behaviour of the terminal in other cases is not specified. The bounding box semantic is described in more detail in Subclause 9.2.13.

Example: This example illustrates a means of avoiding any 2D objects of a group to be, simultaneously, in any other one by ROUTEing the children of the first to the **removeChildren** eventIn of the second:

```
DEF GSrc Group2D {
    children [
     …
    ]
}
DEF GDst Group2D {
     children [
     …
    ]
}
ROUTE GSrc.children  TO GDst.removeChildren
```

9.5.2.2.8 Image2D

9.5.2.2.8.1 Semantic Table

Image2D {

exposedField	MFString	**url**	NULL

}

9.5.2.2.8.2 Main Functionality

This node includes an image in its native size, transmitted in a stream, in a 2D scene. It is different from an **ImageTexture** image in that the image is not scaled to fit the underlying geometry, nor is it texture mapped. The **Image2D** node is a form of geometry node with implicit geometric and visual properties. The implied geometry of this node is defined by the non-transparent pixels of the image and geometry sensors shall respond to this implicit geometry. It is positioned according to its top-left corner and shall not be subjected to transforms described by parent nodes, except for tranlsations.

9.5.2.2.8.3 Detailed Semantics

The **url** field specifies the data source to be used (see 9.2.7.1).

9.5.2.2.9 IndexedFaceSet2D

9.5.2.2.9.1 Semantic Table

```
IndexedFaceSet2D {
    eventIn        MFInt32   set_colorIndex      NULL
    eventIn        MFInt32   set_coordIndex      NULL
    eventIn        MFInt32   set_texCoordIndex   NULL
    exposedField   SFNode    color               NULL
    exposedField   SFNode    coord               NULL
    exposedField   SFNode    texCoord            NULL
    field          MFInt32   colorIndex          NULL
    field          SFBool    colorPerVertex      TRUE
    field          SFBool    convex              TRUE
    field          MFInt32   coordIndex          NULL
    field          MFInt32   texCoordIndex       NULL
}
```

9.5.2.2.9.2 Main Functionality

The **IndexedFaceSet2D** node is the 2D equivalent of the **IndexedFaceSet** node as defined in ISO/IEC 14772-1, Section 6.23. The **IndexedFaceSet2D** node represents a 2D shape formed by constructing 2D faces (polygons) from 2D vertices (points) specified in the **coord** field. The **coord** field contains a Coordinate2D node that defines the 2D vertices, referenced by the **coordIndex** field. The faces of an **IndexedFaceSet2D** node shall not overlap each other.

9.5.2.2.9.3 Detailed Semantics

The detailed semantics are described in ISO/IEC 14772-1, Subclause 6.23, restricted to the 2D case, and with the additional differences described herein.

If the **texCoord** field is NULL, a default texture coordinate mapping is calculated using the local 2D coordinate system bounding box of the 2D shape, as follows. The X dimension of the bounding box defines the S coordinates, and the Y dimension defines the T coordinates. The value of the S coordinate ranges from 0 to 1, from the left end of the bounding box to the right end. The value of the T coordinate ranges from 0 to 1, from the lower end of the bounding box to the top end. Figure 9-10 illustrates the default texture mapping coordinates for a simple **IndexedFaceSet2D** shape consisting of a single polygonal face.

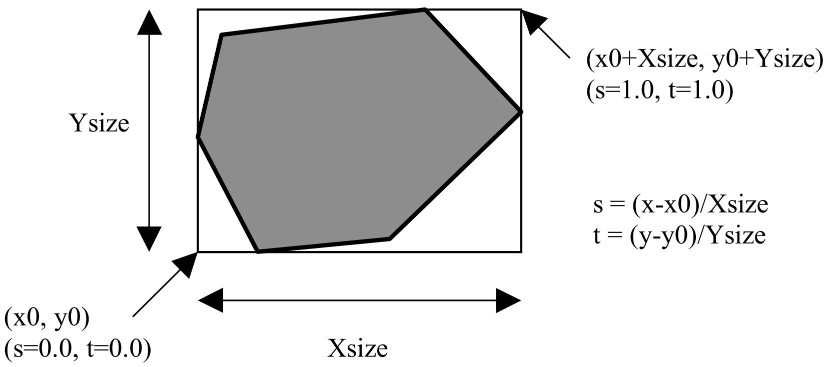


Figure 9-10: IndexedFaceSet2D default texture mapping coordinates for a simple shape

9.5.2.2.10 IndexedLineSet2D

9.5.2.2.10.1 Semantic Table

IndexedLineSet2D {

eventIn	MFInt32	**set_colorIndex**	NULL
eventIn	MFInt32	**set_coordIndex**	NULL
exposedField	SFNode	**color**	NULL
exposedField	SFNode	**coord**	NULL
field	MFInt32	**colorIndex**	NULL
field	SFBool	**colorPerVertex**	TRUE
field	MFInt32	**coordIndex**	NULL

}

9.5.2.2.10.2 Main Functionality

The **IndexedLineSet** node specifies a collection of lines or polygons (depending on the properties2D node).

9.5.2.2.10.3 Detailed Semantics

The **coord** field lists the vertices of the lines. When **coordIndex** is empty, the order of vertices shall be assumed to be sequential in the **coord** field. Otherwise, the **coordIndex** field determines the ordering of the vertices, with an index of -1 representing an end to the current polyline.

If the **color field** is not NULL, it shall contain a **Color** node, and the colors are applied to the line(s) as follows with **IndexedLineSet**.

9.5.2.2.11 Inline2D

9.5.2.2.11.1 Semantic Table

Inline2D {

exposedField	MFString	**url**	NULL
field	SFVec2f	**bboxCenter**	0, 0
field	SFVec2f	**bboxSize**	-1, -1

}

9.5.2.2.11.2 Main Functionality

Inline2D allows the inclusion of a 2D scene from an external source in the current 2D scene graph.

9.5.2.2.11.3 Detailed Semantics

The **url** field specifies the data source to be used (see Subclause 9.2.7.1). The external source must contain a valid 2D BIFS scene, and may include BIFS-Commands and BIFS-Anim frames.

The **bboxCenter** and **bboxSize** semantics are specified in Subclause 9.2.13

9.5.2.2.12 Layout

9.5.2.2.12.1 Semantic Table

Layout {

exposedField	MFNode	**children**	NULL
exposedField	SFBool	**wrap**	FALSE
exposedField	SFVec2f	**size**	-1, -1
exposedField	SFBool	**horizontal**	TRUE
exposedField	MFString	**justify**	["BEGIN"]
exposedField	SFBool	**leftToRight**	TRUE

```
    exposedField    SFBool     topToBottom       TRUE
    exposedField    SFFloat    spacing           1
    exposedField    SFBool     smoothScroll      FALSE
    exposedField    SFBool     loop              FALSE
    exposedField    SFBool     scrollVertical    TRUE
    exposedField    SFFloat    scrollRate        0
    eventIn         MFNode     addChildren       NULL
    eventIn         MFNode     removeChildren    NULL
}
```

9.5.2.2.12.2 Main functionality

The **Layout** node specifies the placement (layout) of its children in various alignment modes as specified, for text children, by their **FontStyle** fields, and for non-text children by the fields **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** present in this node. It also includes the ability to scroll its children horizontally or vertically.

9.5.2.2.12.3 Detailed Semantics

The **children** field shall specify a list of nodes that are to be arranged. Note that the children's position is implicit and that order is important.

The **wrap** field specifies whether children are allowed to wrap to the next row (or column in vertical alignment cases) after the edge of the layout frame is reached. If **wrap** is set to TRUE, children that would be positioned across or past the frame boundary are wrapped (vertically or horizontally) to the next row or column. If **wrap** is set to FALSE, children are placed in a single row or column that is clipped if it is larger than the layout.

When **wrap** is TRUE, if text objects larger than the layout frame need to be placed, these texts shall be broken down into smaller-than-the-layout pieces. The preferred places for breaking a text are spaces, tabs, hyphens, carriage returns and line feeds. When there is no such character in the texts to be broken, the texts shall be broken at the last character that is entirely placed in the layout frame.

The **size** field specifies the width and height of the layout frame.

The **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** fields have the same meaning as in the **FontStyle** node. See ISO/IEC 14772-1, Subclause 6.20, for complete semantics.

The **scrollRate** field specifies the scroll rate in meters per second. When scrollRate is zero, then there is no scrolling and the remaining scroll-related fields are ignored.

The **smoothScroll** field selects between smooth and line-by-line/character-by-character scrolling of children. When TRUE, smooth scroll is applied.

The **loop** field specifies continuous looping of children when set to TRUE. When **loop** is FALSE, child nodes that have scrolled out of the scroll layout frame will be deleted. When **loop** is TRUE, then the set of children scrolls continuously, wrapping around when they have scrolled out of the layout area. If the set of children is smaller than the layout area, some empty space will be scrolled with the children. If the set of children is bigger than the layout area, then only some of the children will be displayed at any point in time. When **scrollVertical** is TRUE and **loop** is TRUE and **scrollRate** is negative (top-to-bottom scrolling), then the bottom-most object will reappear on top of the layout frame as soon as the top-most object has scrolled entirely into the layout frame.

The **scrollVertical** field specifies whether the scrolling is done vertically or horizontally. When set to TRUE, the scrolling rate shall be understood as a vertical scrolling rate, and a positive rate shall mean scrolling to the top. When set to FALSE, the scrolling rate shall be understood as a horizontal scrolling rate, and a positive rate shall mean scrolling to the right.

Objects are placed one by one, in the order they are given in the children list. Text objects are placed according to the **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** fields of their **FontStyle** node. Other objects are placed according to the same fields of the **Layout** node. The reference point for the placement of an object is the reference point as left by the placement of the previous object in the list.

In the case of vertical alignment, objects may be placed with respect to their top, bottom, center or baseline. The baseline of non-text objects is the same as their bottom.

Spacing shall be coherent only within sequences of objects with the same orientation (same value of **horizontal** field). The notions of top edge, bottom edge, base line, vertical center, left edge, right edge, horizontal center, line height and row width shall have a single meaning over coherent sequences of objects. This means that over a sequence of objects where **horizontal** is TRUE, **topToBottom** is TRUE and **spacing** has the same value, then:

- the vertical size of the lines is computed as follows:
  - **maxAscent** is the maximum of the ascent on all text objects.
  - **maxDescent** is the maximum of the descent on all text objects.
  - **maxHeight** is the maximum height of non-text objects.
  - If the minor mode in the **justify** field of the layout is “FIRST” (baseline alignment), then the non-text objects shall be aligned on the baseline, which means the vertical size of the line is:
    size = **max(maxAscent, maxHeight) +maxDescent**
  - If the minor mode in the justify field of the layout is anything else, then the non-text objects shall be aligned with respect to the top, bottom or center, which means the size of the line is:
    size = **max(maxAscent+maxDescent, maxHeight)**
    - the first line is placed with its top edge flush to the top edge of the layout; the base line is placed **maxAscent** units lower, and the bottom edge is placed **maxDescent** units lower; the center line is in the middle between the top and bottom edges; the top edge of following lines are placed at regular intervals of value spacing × size.

The other cases can be inferred from the above description. When the orientation is vertical, then the baseline, ascent and descent are not useful for the computation of the width of the rows. All objects have only a width. Column size is the maximum width over all objects.

Example:

If**wrap** is FALSE:

If **horizontal** is TRUE, then objects are placed in a single line. The layout direction is given by the **leftToRight** field. Horizontal alignment in the row is done according to the first argument in **justify** (major mode = flush left, flush right, centered), and vertical alignment is done according to the second argument in **justify** (minor mode = flush top, flush bottom, flush baseline, centered). The **topToBottom** field is meaningless in this configuration.

If **horizontal** is FALSE, then objects are placed in a single column. The layout direction is given by the **topToBottom** field. Vertical alignment in the column is done according to the first argument in **justify** (major mode), and horizontal alignment is done according to the second argument in justify (minor mode).

If**wrap** is TRUE:

If **horizontal** is TRUE, then objects are placed in multiple lines. The layout direction is given by the **leftToRight** field. The wrapping direction is given by the **topToBottom** field. Horizontal alignment in the lines is done according to the first argument in **justify** (major mode), and vertical alignment is done according to the second argument in **justify** (minor mode).

If **horizontal** is FALSE, then objects are placed in multiple column. The layout direction is given by the **topToBottom** field. The wrapping direction is given by the **leftToRight** field. Vertical alignment in the columns is done according to the first argument in **justify** (major mode), and horizontal alignment is done according to the second argument in **justify** (minor mode).

If**scrollRate** is 0, then the **Layout** is static and positions change only when children are modified.

If **scrollRate** is non zero, then the position of the children is updated according to the values of **scrollVertical**, **scrollRate**, **smoothScroll** and **loop**.

If**scrollVertical** is TRUE:

If**scrollRate** is positive, then the scrolling direction is left-to-right, and vice-versa.

If**scrollVertical** is FALSE:

If**scrollRate** is positive, then the scrolling direction is bottom-to-top, and vice-versa.

9.5.2.2.13 LineProperties

9.5.2.2.13.1 Semantic Table

```
LineProperties {
    exposedField   SFColor    lineColor    0, 0, 0
    exposedField   SFInt32    lineStyle    0
    exposedField   SFFloat    width        1
}
```

9.5.2.2.13.2 Main Functionality

The **LineProperties** node specifies line parameters used in 2D and 3D rendering.

9.5.2.2.13.3 Detailed Semantics

The **lineColor** field determines the color with which to draw the lines and outlines of 2D geometries.

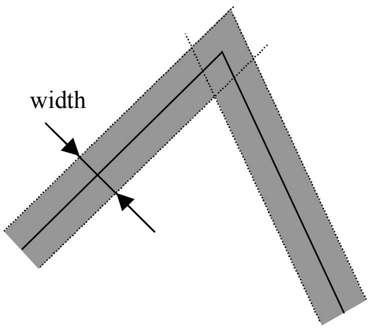
The **lineStyle** field contains the number of the line style to apply to lines. The allowed values are:

lineStyle	Description
0	solid
1	dash
2	dot
3	dash-dot
4	dash-dash-dot
5	dash-dot-dot

The terminal shall draw each line style in a manner that is distiguishable from each other line style.

The **width** field determines the width, in the local coordinate system, of rendered lines. The apparent width depends on the local transformation.

The cap and join style to be used are: the wide lines should end with a square form flush with the end of the lines and the join style is described by the following diagram:



9.5.2.2.14 Material2D

9.5.2.2.14.1 Semantic Table

```
Material2D {
    exposedField   SFColor    emissiveColor    0.8, 0.8, 0.8
    exposedField   SFBool     filled           FALSE
    exposedField   SFNode     lineProps        NULL
    exposedField   SFFloat    transparency     0
}
```

9.5.2.2.14.2 Main Functionality

The **Material2D** node determines the characteristics of a rendered **Shape2D**. This node only appears inside an **Appearance** field, which only appears inside a **Shape2D** node.

9.5.2.2.14.3 Detailed Semantics

The **emissiveColor** field specifies the color of the **Shape2D**.

The **filled** field determines if rendered nodes are filled or drawn using lines. This field affects **IndexedFaceSet2D**, **Circle** and **Rectangle**.

The **lineProps** field contains information about line rendering in the form of a **LineProperties** node. If the field is null the line properties take on a default behaviour identical to the default settings of the **LineProperties** node (see Subclause 9.5.2.2.13) for more information.

The **transparency** field specifies the transparency of the **Shape2D**.

9.5.2.2.15 PlaneSensor2D

9.5.2.2.15.1 Semantic Table

PlaneSensor2D {

exposedField	SFBool	**autoOffset**	TRUE
exposedField	SFBool	**enabled**	TRUE
exposedField	SFVec2f	**maxPosition**	0, 0
exposedField	SFVec2f	**minPosition**	0, 0
exposedField	SFVec2f	**offset**	0, 0
eventOut	SFVec2f	**trackPoint_changed**	

}

9.5.2.2.15.2 Main Functionality

This sensor detects pointer device dragging and enables the dragging of objects on the 2D rendering plane.

9.5.2.2.15.3 Detailed Semantics

The semantic is a restricted case for 2D of the **PlaneSensor** as defined in ISO/IEC 14772-1.

9.5.2.2.16 PointSet2D

9.5.2.2.16.1 Semantic Table

PointSet2D {

exposedField	SFNode	**color**	NULL
exposedField	SFNode	**coord**	NULL

}

9.5.2.2.16.2 Main Functionality

This is a 2D equivalent of the **PointSet** node.

9.5.2.2.16.3 Detailed Semantics

The semantics are the 2D restriction of the **PointSet** node as defined in ISO/IEC 14772-1.

9.5.2.2.17 Position2DInterpolator

9.5.2.2.17.1 Semantic Table

```
Position2DInterpolator {
    eventIn        SFFloat     set_fraction         NULL
    exposedField   MFFloat     key                  NULL
    exposedField   MFVec2f     keyValue             NULL
    eventOut       SFVec2f     value_changed
}
```

9.5.2.2.17.2 Main Functionality

This is a 2D equivalent of the **PositionInterpolator** node.

9.5.2.2.17.3 Detailed Semantics

The semantics are the 2D restriction of the **PositionInterpolator** node as defined in ISO/IEC 14772-1.

9.5.2.2.18 Proximity2DSensor

9.5.2.2.18.1 Semantic Table

```
Proximity2DSensor {
    exposedField   SFVec2f     center               0, 0
    exposedField   SFVec2f     size                 0, 0
    exposedField   SFBool      enabled              TRUE
    eventOut       SFBool      isActive
    eventOut       SFVec2f     position_changed
    eventOut       SFFloat     orientation_changed
    eventOut       SFTime      enterTime
    eventOut       SFTime      exitTime
}
```

9.5.2.2.18.2 Main Functionality

This is the 2D equivalent of the **ProximitySensor** node.

9.5.2.2.18.3 Detailed Semantics

The semantics are the 2D restriction of the **PointSet** node as defined in ISO/IEC 14772-1.

9.5.2.2.19 Rectangle

9.5.2.2.19.1 Semantic Table

```
Rectangle {
    exposedField   SFVec2f     size                 2, 2
}
```

9.5.2.2.19.2 Main Functionality

This node renders a rectangle.

9.5.2.2.19.3 Detailed Semantics

The **size** field specifies the horizontal and vertical size of the rendered rectangle.

9.5.2.2.20 Sound2D

9.5.2.2.20.1 Semantic table

Sound2D {

exposedField	SFFloat	**intensity**	1
exposedField	SFVec2f	**location**	0,0
exposedField	SFNode	**source**	NULL
field	SFBool	**spatialize**	TRUE

}

9.5.2.2.20.2 Main functionality

The **Sound2D** node relates an audio BIFS subtree to the other parts of a 2D audiovisual scene. By using this node, sound may be attached to a group of visual nodes. By using the functionality of the audio BIFS nodes, sounds in an audio scene may be filtered and mixed before being spatially composed into the scene.

9.5.2.2.20.3 Detailed semantics

The **intensity** field adjusts the loudness of the sound. Its value ranges from 0.0 to 1.0, and this value specifies a factor that is used during the playback of the sound.

The **location** field specifies the location of the sound in the 2D scene.

The **source** field connects the audio source to the **Sound2D** node.

The **spatialize** field specifies whether the sound should be spatialized on the 2D screen. If this flag is set, the sound shall be spatialized with the maximum sophistication possible. The 2D sound is spatialized assuming a distance of one meter between the user and a 2D scene of size 2m x 1.5m, giving the minimum and maximum azimuth angles of –45° and +45°, and the minimum and maximum elevation angles of -37° and +37 °.

The same rules for multichannel audio spatialization apply to the **Sound2D** node as to the **Sound** (3D) node. Using the **phaseGroup** flag in the **AudioSource** node it is possible to determine whether the channels of the source sound contain important phase relations, and that spatialization at the terminal should not be performed.

As with the visual objects in the scene (and for the **Sound** node), the **Sound2D** node may be included as a child or descendant of any of the grouping or transform nodes. For each of these nodes, the sound semantics are as follows.

Affine transformations presented in the grouping and transform nodes affect the apparent spatialization position of spatialized sound.

If a transform node has multiple **Sound2D** nodes as descendants, then they are combined for presentation as follows:

- Ambient sounds, i.e., sounds with the spatialization flag set to zero, are linearly combined channel-by-channel for presentation.
- For sounds with the spatialize flag=1, the channels are first summed and the resulting monophonic sound is spatialized according to the location in the scene.

9.5.2.2.21 Switch2D

9.5.2.2.21.1 Semantic Table

Switch2D {

exposedField	MFNode	**choice**	NULL
exposedField	SFInt32	**whichChoice**	-1

}

9.5.2.2.21.2 Main functionality

The **Switch2D** grouping node traverses zero or one of the 2D nodes specified in the **choice** field. All nodes contained in a **Switch2D** continue to receive and send events regardless of the choice of the traversed one.

9.5.2.2.21.3 Detailed Semantics

The **choice** field specifies the list switchable nodes.

The **whichChoice** field specifies the index of the child to traverse, with the first child having index 0. If **whichChoice** is less than zero or greater than the number of nodes in the **choice** field, nothing is chosen.

9.5.2.2.22 Transform2D

9.5.2.2.22.1 Semantic Table

```
Transform2D {
    eventIn         MFNode     addChildren          NULL
    eventIn         MFNode     removeChildren       NULL
    exposedField    SFVec2f    center               0, 0
    exposedField    MFNode     children             NULL
    exposedField    SFFloat    rotationAngle        0
    exposedField    SFVec2f    scale                1, 1
    exposedField    SFFloat    scaleOrientation     0
    exposedField    SFFloat    drawingOrder         0
    exposedField    SFVec2f    translation          0, 0
    field           SFVec2f    bboxCenter           0, 0
    field           SFVec2f    bboxSize             -1, -1
}
```

9.5.2.2.22.2 Main Functionality

The **Transform2D** node allows the translation, rotation and scaling of its 2D children objects.

9.5.2.2.22.3 Detailed Semantics

The **bboxCenter** and **bboxSize** semantics are specified in Subclause 9.2.13.

The **rotation** field specifies a rotation of the child objects, in radians, which occurs about the point specified by **center**.

The **scale** field specifies a 2D scaling of the child objects. The scaling operation takes place following a rotation of the 2D coordinate system that is specified, in radians, by the **scaleOrientation** field. The rotation of the co-ordinate system is notional and purely for the purpose of applying the scaling and is undone before any further actions are performed. No permanent rotation of the co-ordinate system is implied.

The **translation** field specifies a 2D vector which translates the child objects.

The scaling, rotation and translation are applied in the following order: scale, rotate, translate.

The **drawingOrder** field specifies the order in which this node's children are drawn *with respect to other objects in the scene* (see 9.2.15) . When a **Transform2D** node has more than one child node, its children are drawn in order. The exception to this rule occurs when one or more child node has an explicit **drawingOrder**. In this case, the explicit **drawingOrder** is respected for that child node without affecting the implicit drawing order of its siblings.

The **children** field contains a list of zero or more children nodes which are grouped by the **Transform2D** node.

The **addChildren** and **removeChildren eventIns** are used to add or remove child nodes from the **children** field of the node. Children are added to the end of the list of children and special note should be taken of the implications of this for implicit drawing orders.

9.5.2.2.23 VideoObject2D

9.5.2.2.23.1 Semantic Table

```
VideoObject2D {
    exposedField    SFBool     loop                 FALSE
    exposedField    SFFloat    speed                1
```

```
    exposedField    SFTime      startTime           0
    exposedField    SFTime      stopTime            0
    exposedField    MFString    url                 NULL
    eventOut        SFFloat     duration_changed
    eventOut        SFBool      isActive
}
```

9.5.2.2.23.2 Main Functionality

The **VideoObject2D** node includes a video sequence using its natural size into a 2D scene. It is different from **MovieTexture** in that the video sequence is not scaled to fit the underlying geometry, nor is it texture mapped. The **VideoObject2D** node is a form of geometry node with implicit geometric and visual properties. The implied geometry of this node is defined by the non-transparent pixels of the video sequence and geometry sensors shall respond to this implicit geometry. It is positioned according to its top-left corner and shall not be subjected to transforms described by parent nodes, except for tranlsations.

9.5.2.2.23.3 Detailed Semantics

As soon as the movie is started, a **duration_changed** eventOut is sent. This indicates the duration of the video object in seconds. This eventOut value can be read to determine the duration of a video object. A value of "-1" implies the video object has not yet loaded or the value is unavailable.

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **VideoObject2D** node, are described in Subclause 9.2.13. The cycle of a **VideoObject2D** node is the length of time in seconds for one playing of the video object at the specified speed.

The **speed** exposedField indicates how fast the movie shall be played. A speed of 2 indicates the movie plays twice as fast. The **duration_changed** eventOut is not affected by the **speed** exposedField. **set_speed** events are ignored while the movie is playing. A negative speed implies that the movie will play backwards. For streaming media, negative values of **speed** are not implementable and shall be ignored.

If a **VideoObject2D** node is inactive when the video object is first loaded, frame or VOP 0 of the video object is displayed if **speed** is non-negative or the last frame or VOP of the video object is shown if **speed** is negative. A **VideoObject2D** node shall display frame or VOP 0 if **speed** = 0. For positive values of **speed**, an active **VideoObject2D** node displays the frame or VOP at video object time t as follows (i.e., in the video object's local time system with frame or VOP 0 at time 0 with speed = 1):

t = (now - startTime) modulo (duration/speed)

If**speed** is negative, the **VideoObject2D** node displays the frame or VOP at movie time:

t = duration - ((now - startTime) modulo ABS(duration/speed))

When a **VideoObject2D** node becomes inactive, the frame or VOP corresponding to the time at which the **VideoObject2D** became inactive will remain visible.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1).

9.5.3 3D Nodes

9.5.3.1 3D Nodes Overview

The 3D nodes are those nodes which may be used in 3D scenes.

9.5.3.2 3D Native Nodes

The following nodes are specific to this Final Committee Draft of International Standard.

9.5.3.2.1 ListeningPoint

9.5.3.2.1.1 Semantic Table

ListeningPoint {

eventIn	SFBool	**set_bind**	NULL
exposedField	SFBool	**jump**	TRUE
exposedField	SFRotation	**orientation**	0, 0, 1, 0
exposedField	SFVec3f	**position**	0, 0, 10
field	SFString	**description**	""
eventOut	SFTime	**bindTime**	
eventOut	SFBool	**isBound**	

}

9.5.3.2.1.2 Main Functionality

The **ListeningPoint** node specifies the reference position and orientation for spatial audio presentation. If there is no **ListeningPoint** given in a scene, the apparent listener position is slaved to the active **ViewPoint**.

9.5.3.2.1.3 Detailed Semantics

The semantics are identical to those of the **Viewpoint** node (see Subclause 9.5.3.3.30).

9.5.3.2.2 Face

9.5.3.2.2.1 Semantic Table

Face {

exposedField	SFNode	**fit**	NULL
exposedField	SFNode	**fdp**	NULL
exposedField	SFNode	**fap**	NULL
exposedField	MFString	**url**	NULL
exposedField	MFNode	**renderedFace**	NULL

}

9.5.3.2.2.2 Main Functionality

The **Face** node organizes the definition and animation of a face. The **FAP** field shall be always specified in a **Face** node; the **FDP** field, defining the particular look of a face by means of downloading the position of face definition points or an entire model, is optional. If the **FDP** field is not specified, the default face model of the decoder is used. The **FIT** field, when specified, allows a set of Facial Animation Parameters (FAPs) to be defined in terms of another set of FAPs.

The **url** field specifies the data source to be used (see 9.2.7.1).

9.5.3.2.2.3 Detailed Semantics

fit specifies the FIT node. When this field is non-null, the decoder should use FIT to compute the maximal set of FAPs before using the FAPs to compute the mesh. **fdp** contains an FDP node. **fap** contains an FAP node.

The **url** field specifies the data source to be used (see Subclause 9.2.7.1)

renderedFace is the scene graph of the face after it is rendered (all FAP's applied)

9.5.3.2.3 FAP

9.5.3.2.3.1 Semantic Table

FAP {

exposedField	SFNode	**viseme**	0
exposedField	SFNode	**expression**	0
exposedField	SFInt32	**open_jaw**	+I
exposedField	SFInt32	**lower_t_midlip**	+I
exposedField	SFInt32	**raise_b_midlip**	+I
exposedField	SFInt32	**stretch_l_corner**	+I

exposedField	SFInt32	**stretch_r_corner**	+I
exposedField	SFInt32	**lower_t_lip_lm**	+I
exposedField	SFInt32	**lower_t_lip_rm**	+I
exposedField	SFInt32	**lower_b_lip_lm**	+I
exposedField	SFInt32	**lower_b_lip_rm**	+I
exposedField	SFInt32	**raise_l_cornerlip**	+I
exposedField	SFInt32	**raise_r_cornerlip**	+I
exposedField	SFInt32	**thrust_jaw**	+I
exposedField	SFInt32	**shift_jaw**	+I
exposedField	SFInt32	**push_b_lip**	+I
exposedField	SFInt32	**push_t_lip**	+I
exposedField	SFInt32	**depress_chin**	+I
exposedField	SFInt32	**close_t_l_eyelid**	+I
exposedField	SFInt32	**close_t_r_eyelid**	+I
exposedField	SFInt32	**close_b_l_eyelid**	+I
exposedField	SFInt32	**close_b_r_eyelid**	+I
exposedField	SFInt32	**yaw_l_eyeball**	+I
exposedField	SFInt32	**yaw_r_eyeball**	+I
exposedField	SFInt32	**pitch_l_eyeball**	+I
exposedField	SFInt32	**pitch_r_eyeball**	+I
exposedField	SFInt32	**thrust_l_eyeball**	+I
exposedField	SFInt32	**thrust_r_eyeball**	+I
exposedField	SFInt32	**dilate_l_pupil**	+I
exposedField	SFInt32	**dilate_r_pupil**	+I
exposedField	SFInt32	**raise_l_i_eyebrow**	+I
exposedField	SFInt32	**raise_r_i_eyebrow**	+I
exposedField	SFInt32	**raise_l_m_eyebrow**	+I
exposedField	SFInt32	**raise_r_m_eyebrow**	+I
exposedField	SFInt32	**raise_l_o_eyebrow**	+I
exposedField	SFInt32	**raise_r_o_eyebrow**	+I
exposedField	SFInt32	**squeeze_l_eyebrow**	+I
exposedField	SFInt32	**squeeze_r_eyebrow**	+I
exposedField	SFInt32	**puff_l_cheek**	+I
exposedField	SFInt32	**puff_r_cheek**	+I
exposedField	SFInt32	**lift_l_cheek**	+I
exposedField	SFInt32	**lift_r_cheek**	+I
exposedField	SFInt32	**shift_tongue_tip**	+I
exposedField	SFInt32	**raise_tongue_tip**	+I
exposedField	SFInt32	**thrust_tongue_tip**	+I
exposedField	SFInt32	**raise_tongue**	+I
exposedField	SFInt32	**tongue_roll**	+I
exposedField	SFInt32	**head_pitch**	+I
exposedField	SFInt32	**head_yaw**	+I
exposedField	SFInt32	**head_roll**	+I
exposedField	SFInt32	**lower_t_midlip_o**	+I
exposedField	SFInt32	**raise_b_midlip_o**	+I
exposedField	SFInt32	**stretch_l_cornerlip**	+I
exposedField	SFInt32	**stretch_r_cornerlip_o**	+I
exposedField	SFInt32	**lower_t_lip_lm_o**	+I
exposedField	SFInt32	**lower_t_lip_rm_o**	+I
exposedField	SFInt32	**raise_b_lip_lm_o**	+I
exposedField	SFInt32	**raise_b_lip_rm_o**	+I
exposedField	SFInt32	**raise_l_cornerlip_o**	+I
exposedField	SFInt32	**raise_r_cornerlip_o**	+I
exposedField	SFInt32	**stretch_l_nose**	+I
exposedField	SFInt32	**stretch_r_nose**	+I
exposedField	SFInt32	**raise_nose**	+I
exposedField	SFInt32	**bend_nose**	+I
exposedField	SFInt32	**raise_l_ear**	+I

```
    exposedField    SFInt32    raise_r_ear    +I
    exposedField    SFInt32    pull_l_ear     +I
    exposedField    SFInt32    pull_r_ear     +I
}
```

9.5.3.2.3.2 Main Functionality

This node defines the current look of the face by means of expressions and FAPs and gives a hint to TTS controlled systems on which viseme to use. For a definition of the parameters see Part 2 of this Final Committee Draft of International Standard.

9.5.3.2.3.3 Detailed Semantics

viseme Contains a Viseme node.

expression Contains and Expression node.

open_jaw, …, pull_r_ear The semantics for these parameters are described in the ISO/IEC FCD 14496-2, Annex C and in particular in Table 12-1.

A FAP of value +I is assumed to be uninitialized.

9.5.3.2.4 Viseme

9.5.3.2.4.1 Semantic Table

```
Viseme {
    field    SFInt32    viseme_select1    0
    field    SFInt32    viseme_select2    0
    field    SFInt32    viseme_blend      0
    field    SFBool     viseme_def        0
}
```

9.5.3.2.4.2 Main Functionality

The **Viseme** node defines a blend of two visemes from a standard set of 14 visemes as defined in Part 2 of this Final Committee Draft of International Standard, Table 12-5.

9.5.3.2.4.3 Detailed Semantics

viseme_select1 Specifies viseme 1.

viseme_select2 Specifies viseme 2.

viseme_blend Specifies the blend of the two visemes.

viseme_def If viseme_def is set, current FAPs are used to define a viseme and store it.

9.5.3.2.5 Expression

9.5.3.2.5.1 Semantic Table

```
Expression {
    field    SFInt32    expression_select1      0
    field    SFInt32    expression_intensity1   0
    field    SFInt32    expression_select2      0
    field    SFInt32    expression_intensity2   0
    field    SFBool     init_face               0
    field    SFBool     expression_def          0
```

}

9.5.3.2.5.2 Main Functionality

The **Expression** node is used to define the expression of the face as a combination of two expressions out of the standard set of expressions defined in Part 2 of this Final Committee Draft of International Standard, Table 12-3.

9.5.3.2.5.3 Detailed Semantics

expression_select1	specifies expression 1.
expression_intensity1	specifies intensity for expression 1.
expression_select2	specifies expression 2.
expression_intensity2	specifies intensity for expression 2.
init_face	If init_face is set, neutral face may be modified before applying FAPs 1 and 3-68.
expression_def	If expression_def is set, current FAPs are used to define an expression and store it.

9.5.3.2.6 FIT

9.5.3.2.6.1 Semantic Table

```
FIT {
    exposedField   MFInt32   FAPs               NULL
    exposedField   MFInt32   graph              NULL
    exposedField   MFInt32   numeratorTerms     NULL
    exposedField   MFInt32   denominatorTerms   NULL
    exposedField   MFInt32   numeratorExp       NULL
    exposedField   MFInt32   denominatorExp     NULL
    exposedField   MFInt32   numeratorImpulse   NULL
    exposedField   MFFloat   numeratorCoefs     NULL
    exposedField   MFFloat   denominatorCoefs   NULL
}
```

9.5.3.2.6.2 Main Functionality

The **FIT** node allows a smaller set of FAPs to be sent during a facial animation. This small set can then be used to determine the values of other FAPs, using a rational polynomial mapping between parameters. For example, the top inner lip FAPs can be sent and then used to determine the top outer lip FAPs. Another example is that only viseme and/or expression FAPs are sent to drive the face. In this case, low-level FAPs are interpolated from these two high-level FAPs. In FIT, rational polynomials are used to specify interpolation functions.

To make the scheme general, sets of FAPs are specified, along with a FAP Interpolation Graph (FIG) between the sets that specifies which sets are used to determine which other sets. The FIG is a graph with directed links. Each node contains a set of FAPs. Each link from a parent node to a child node indicates that the FAPs in the child node can be interpolated from the parent node. Expression (FAP#1) or viseme (FAP #2) and their subfields shall not be interpolated from other FAPs.

In a FIG, a FAP may appear in several nodes, and a node may have multiple parents. For a node that has multiple parent nodes, the parent nodes are ordered as 1st parent node, 2nd parent node, etc. During the interpolation process, if this child node needs to be interpolated, it is first interpolated from 1st parent node if all FAPs in that parent node are available. Otherwise, it is interpolated from 2nd parent node, and so on.

An example of FIG is shown in Figure 9-11. Each node has a nodeID. The numerical label on each incoming link indicates the order of these links.

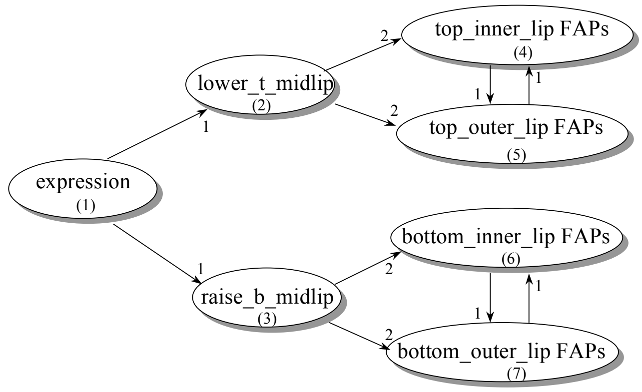


Figure 9-11: A FIG example.

The interpolation process based on the FAP interpolation graph is described using pseudo C code as follows:

```
do {
   interpolation_count = 0;
   for (all Node_i) {   // from Node_1 to Node_N
      for (ordered Node_i's parent Node_k) {
         if (FAPs in Node_i need interpolation and
                  FAPs in Node_k have been interpolated or are available) {
            interpolate Node_i from Node_k;  //using interpolation function
                                             // table here
            interpolation_count ++;
            break;
         }
      }
   }
} while (interpolation_count != 0);
```

Each directed link in a FIG is a set of interpolation functions. Suppose $F_1, F_2, \ldots, F_n$ are the FAPs in a parent set and $f_1, f_2, \ldots, f_m$ are the FAPs in a child set.
Then, there are m interpolation functions denoted as:

$f_1 = I_1(F_1, F_2, \ldots, F_n)$
$f_2 = I_2(F_1, F_2, \ldots, F_n)$
…
$f_m = I_m(F_1, F_2, \ldots, F_n)$

Each interpolation function I_k () is in a rational polynomial form if the parent node does not contain viseme FAP or expression FAP.

$$I(F_1, F_2, \ldots, F_n) = \sum_{i=0}^{K-1} (c_i \prod_{j=1}^{n} F_j^{l_{ij}}) \Big/ \sum_{i=0}^{P-1} (b_i \prod_{j=1}^{n} F_j^{m_{ij}}) \tag{1}$$

otherwise, an impulse function is added to each numerator polynomial term to allow selection of expression or viseme.

$$I(F_1, F_2, \ldots, F_n) = \sum_{i=0}^{K-1} \delta(F_{s_i} - a_i)(c_i \prod_{j=1}^{n} F_j^{l_{ij}}) \Big/ \sum_{i=0}^{P-1} (b_i \prod_{j=1}^{n} F_j^{m_{ij}}) \tag{2}$$

In both equations, K and P are the numbers of polynomial products, c_i and b_i are the coefficient of the ith product. l_{ij} and m_{ij} are the power of F_j in the ith product. An impulse function equals 1 when $F_{s_i} = a_i$, otherwise, equals 0. F_{s_i} can only be viseme_select1, viseme_select2, expression_select1, and expression_select2. a_i is an integer that ranges from 0 to 6 when F_{s_i} is expression_select1 or expression_select2, ranges 0 to 14 when F_{s_i} is viseme_select1 or viseme_select2. The encoder should send an interpolation function table which contains K , P , a_i , s_i , c_i , b_i , l_{ij} , m_{ij} to the decoder.

9.5.3.2.6.3 Detailed Semantics

To aid in the explanation below, we assume that there are N different sets of FAPs with index 1 to N, and that each set has n_i, i=1,..,N parameters. We assume that there are L directed links in the FIG and that each link points from the FAP set with index P_i to the FAP set with index C_i, for i = 1, .. , L

FAPs A list of FAP-indices specifying which animation parameters form sets of FAPs. Each set of FAP indices is terminated by a –1. There should be a total of $N + n_1 + n_2 + \ldots + n_N$ numbers in this field, with N of them being –1. FAP#1 to FAP#68 are of indices 1 to 68. Sub-fields of viseme FAP (FAP#1), namely, viseme_select1, viseme_select2, viseme_blend, are of indices from 69 to 71. Sub-fields of expression FAP (FAP#2), namely, expression_select1, expression_select2, expression_intensity1, expression_intensity2 are of indices from 72 to 75. When the parent node contains a viseme FAP, three indices 69,70,71 should be include in the node (but not index 1). When a parent node contains an expression FAP, four indices 72,73,74,75 should be included in the node (but not index 2).

graph A list of pairs of intergers, specifying a directed links between sets of FAPs. The integers refer to the indices of the sets that specified in the **FAPs** field, thus range from 1 to N. When more than one direct link terminates at the same set, that is, when the second value in the pair is repeated, the links have precedence determined by their order in this field. This field should have a total of 2L numbers, corresponding to the directed links between the parents and children in the FIG.

numeratorTerms A list containing the number of terms in the polynomials in the numerators of the rational functions used for interpolating parameter values. Each element in the list corresponds to K in equation 1 above). Each link i (that is, the ith integer pair) in **graph** must have n_{Ci} values specified, one for each child FAP. The order in the **numeratorTerms** list corresponds to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPSs** field. There should be $n_{C1} + n_{C2} + \ldots + n_{CL}$ numbers in this field.

denominatorTerms A list of the number of terms in the polynomials in the denominator of the rational functions controlling the parameter value. . Each element in the list corresponds to P in equation 1. Each link i (that is, the ith integer pair) in **graph** must have n_{Ci} values specified, one for each child FAP. The order in the **denominatorTerms** list corresponds to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs**field. There should be $n_{C1} + n_{C2} + \ldots + n_{CL}$ numbers in this field.

numeratorImpulse A list of impulse functions in the numerator of the rational function for links with viseme or expression FAP in parent node. This list corresponds to the $\delta(F_{s_i} - a_i)$. Each entry in the list is (s_i , a_i).

numeratorExp A list of exponents of the polynomial terms in the numerator of the rational function controlling the parameter value. This list corresponds to l_{ij} . For each child FAP in each link i, n_{Pi}*K values need to be specified. Note that K may be different for each child FAP. The order in the **numeratorExp** list corresponds to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field.

denominatorExp A list of exponents of the polynomial terms in the denominator of the rational function controlling the parameter value. This list corresponds to m_{ij}. For each child FAP in each link i, n_{Pi}*P values need to be specified. Note that P may be different for each child FAP. The order in the **denominatorExp** list corresponds to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field.

numeratorCoefs A list of coefficients of the polynomial terms in the numerator of the rational function controlling the parameter value. This list corresponds to c_i. The list should have *K* terms for each child parameter that appears in a link in the FIG, with the order in **numeratorCoefs** corresponding to the order in **graph** and **FAPs**. Note that *K* is dependent on the polynomial, and is not a fixed constant.

denominatorCoefs A list of coefficients of the polynomial terms in the numerator of the rational function controlling the parameter value. This list corresponds to b_i. The list should have *P* terms for each child parameter that appears in a link in the FIG, with the order in **denominatorCoefs** corresponding to the order in **graph** and **FAPs** Note that *P* is dependent on the polynomial, and is not a fixed constant.

Example: Suppose a FIG contains four nodes and 2 links. Node 1 contains FAP#3, FAP#3, FAP#5. Node 2 contains FAP#6, FAP#7. Node 3 contains an expression FAP, which means contains FAP#72, FAP#73, FAP#74, and FAP#75. Node 4 contains FAP#12 and FAP#17. Two links are from node 1 to node 2, and from node 3 to node 4. For the first link, the interpolation functions are

$$F_6 = (F_3 + 2F_4 + 3F_5 + 4F_3F_4^2)/(5F_5 + 6F_3F_4F_5)$$

$$F_7 = F_4.$$

For the second link, the interpolation functions are

$$F_{12} = \delta(F_{72} - 6)(0.6F_{74}) + \delta(F_{73} - 6)(0.6F_{75})$$

$$F_{17} = \delta(F_{72} - 6)(-1.5F_{74}) + \delta(F_{73} - 6)(-1.5F_{75}).$$

The second link simply says that when the expression is surprise (FAP#72=6 or FAP#73=6), for FAP#12, the value is 0.6 times of expression intensity FAP#74 or FAP#75; for FAP#17, the value is –1.5 tims of FAP#74 or FAP#75.

After the FIT node given below, we explain each field separately.

```
FIT {
   FAPs              [ 3  4  5  -1  6  7  -1  72  73  74  75  -1  12  17  -1]
   graph             [ 1  2  3  4]
   numeratorTerms    [ 4  1  2  2 ]
   denominatorTerms  [2  1  1  1]
   numeratorExp      [1  0  0    0  1  0    0  0  1   1  2  0   0  1  0
                        0  0  1  0   0  0  0  1   0  0  1  0   0  0  0  1 ]
   denominatorExp    [ 0  0  1   1  1  1   0  0  0
                        0  0  0  0   0  0  0  0 ]
   numeratorImpulse  [ 72  6   73  6   72  6   73  6 ]
   numeratorCoefs    [1  2  3  4    1    0.6  0.6   -1.5  -1.5 ]
   denominatorCoefs  [5  6  1  1  1 ]
}
```

- `FAPs [ 3 4 5 -1 6 7 -1 72 73 74 75 -1 12 17 -1]`
  We define four sets of FAPs, the first with FAPs number 3, 4, and 5, the second with FAPs number 6 and 7, the third with FAPs number 72, 73, 74, 75, and the fourth with FAPs number 12, 17.
- `graph [ 1 2 3 4]`
  We will make the first set the parent of the second set, so that FAPs number 6 and 7 will be determined by FAPs 3, 4, and 5. Also, we will make the third set the parent of the fourth set, so that FAPs number 12 and 17 will be determined by FAPs 72, 73, 74, and 75.
- `numeratorTerms [ 4 1 2 2]`

We select the rational functions that define F_6 and F_7 to have 4 and 1 terms in their numerator, respectively. Also, we select the rational functions that define F_{12} and F_{17} to have 2 and 2 terms in their numerator, respectively.

- `denominatorTerms [ 2 1 1 1]`
  We select the rational functions that define F_6 and F_7 to have 2 and 1 terms in their denominator, respectively. Also, we select the rational functions that define F_{12} and F_{17} to both have 1 term in their denominator.
- `numeratorExp [ 1 0 0   0 1 0   0 0 1  1 2 0      0 1 0   0 0 1 0  0 0 0 1  0 0 1 0  0 0 0 1]`
  The numerator we select for the rational function defining F_6 is $F_3 + 2F_4 + 3 F_5 + 4F_3F_4^2$. There are 3 parent FAPs, and 4 terms, leading to 12 exponents for this rational function. For F_7, the numerator is just F_4, so there are three exponents only (one for each FAP). Values for F_{12} and F_{17} are derived in the same way.
- `denominatorExp [ 0 0 1  1 1 1    0 0 0   0 0 0 0  0 0 0 0]`
  The denominator we select for the rational function defining F_6 is $5F_5+ 6F_3F_4F_5$, so there are 3 parent FAPs and 2 terms and hence, 6 exponents for this rational function. For F_7, the denominator is just 1, so there are three exponents only (one for each FAP). Values for F_{12} and F_{17} are derived in the same way.
- `numeratorImpulse [72 6  73 6  72 6  73 6]`
  For the second link, all four numerator polynomial terms contain impulse function $\delta(F_{72} - 6)$ or $\delta(F_{73} - 6)$.
- `numeratorCoefs [ 1 2 3 4  1   0.6 0.6  -1.5 –1.5]`
  There is one coefficient for each term in the numerator of each rational function.
- `denominatorCoefs [ 5 6    1   1 1]`
  There is one coefficient for each term in the denominator of each rational function.

9.5.3.2.7 FDP

9.5.3.2.7.1 Semantic Table

FDP {

exposedField	SFNode	**featurePointsCoord**	NULL
exposedField	SFNode	**textureCoords**	NULL
exposedField	MFNode	**faceDefTables**	NULL
exposedField	MFNode	**faceSceneGraph**	NULL

}

9.5.3.2.8 Main Functionality

The **FDP** node defines the face model to be used at the receiver. Two options are supported:

1. if **faceDefTables** is NULL, calibration information is downloaded, so that the proprietary face of the receiver can be calibrated using facial feature points and optionally the texture information. In this case, the field **featurePointsCoord** has to be set. **featurePointsCoord** contains the coordinates of facial feature points, as defined in ISO/IEC FCD 14496-2, Figure 12-1, corresponding to a neutral face. If a coordinate of a feature point is set to +I, the coordinates of this feature point are to be ignored. **textureCoord**, if set, are used to map a texture on the model calibrated by the feature points. They correspond to the feature points, i.e. each defined feature point must have corresponding texture coordinates. In this case, the **faceSceneGraph** must contain exactly one texture image, and any geometry it might contain is ignored. The decoder interprets the feature points, texture coordinates, and the **faceSceneGraph** in the following way:
   a) Feature points of the decoder's face model are moved to the coordinates of the feature points supplied in **featurePointsCoord**, unless a feature point is to be ignored as explained above.
   b) If **textureCoord** is set, the texture supplied in the **faceSceneGraph** is mapped on the proprietary model. The texture coordinates are derived from the texture coordinates of the feature points supplied in **textureCoords**.
2. a face model as described in the **faceSceneGraph** is downloaded. This face model replaces the proprietary face model in the receiver. The **faceSceneGraph** has to have the face in its neutral position (all FAPs 0). If desired, the **faceSceneGraph** contains the texture maps of the face. The definition of how to modify the **faceSceneGraph** as a function of the FAPs has also to be downloaded to the decoder. This information is described by **faceDefTables** that define how the **faceSceneGraph** has to be modified as a function of each FAP. By means of **faceDefTables**, indexed face sets and transform nodes of the **faceSceneGraph** can be animated.

Since the amplitude of FAPs is defined in units depending on the size of the face model, the field **featurePointsCoord** define the position of facial features on the surface of the face described by **faceSceneGraph**. From the location of these feature points, the decoder computes the units of the FAPs**.**
Generally only two node types in the scenegraph of a downloaded face model are affected by FAPs: **IndexedFaceSet** and **Transform** nodes. If a FAP causes a deformation of an object (e.g. lip stretching), then the coordinate positions in the affected 'IndexedFaceSets' shall be updated. If a FAP causes a movement which can be described with a **Transform** node (e.g. FAP 23, yaw_l_eyeball), then the appropriate fields in this **Transform** node shall be updated. It is assumed that this transform node has its fields rotation, scale, and translation set to neutral if the face is in its neutral position. A unique 'nodeId' must be assigned via the **DEF** statement to all **IndexedFaceSet** and **Transform** nodes which are affected by FAPs so that they can be accessed unambiguously during animation.

9.5.3.2.9 Detailed Semantics

featurePointsCoord contains a **Coordinate** node. Specifies feature points for the calibration of the proprietary face. The coordinates are listed in the 'point' field in the **Coordinate** node in the prescribed order, that a feature point with a lower label is listed before a feature point with a higher label (e.g. feature point 3.14 before feature point 4.1).

textureCoords contains a **Coordinate** node. Specifies texture coordinates for the feature points. The coordinates are listed in the **point** field in the **Coordinate** node in the prescribed order, that a feature point with a lower label is listed before a feature point with a higher label (e.g. feature point 3.14 before feature point 4.1).

faceDefTables contains **faceDefTables** nodes. The behavior of FAPs is defined in this field for the face in **faceSceneGraph**.

faceSceneGraph contains a **Group**node. In case of option 1, this can be used to contain a texture image as explained above. In case of option 2, this is the grouping node for face model rendered in the compositor and has to contain the face model. In this case, the effect of Facial Animation Parameters is defined in the **faceDefTablesfield**.

9.5.3.2.10 FaceDefTables

9.5.3.2.10.1 Semantic Table

```
FaceDefTables {
    field           SFInt32     fapID               0
    field           SFInt32     highLevelSelect     0
    exposedField    MFNode      faceDefMesh         NULL
    exposedField    MFNode      faceDefTransform    NULL
}
```

9.5.3.2.10.2 Main Functionality

Defines the behavior of a facial animation parameter FAP on a downloaded **faceSceneGraph** by specifying displacement vectors for moved vertices inside **IndexedFaceSet** objects as a function of the FAP **fapID** and/or specifying the value of a field of a **Transform** node as a function of FAP **fapID**.

The **FaceDefTables** node is transmitted directly after the BIFS bitstream of the **FDP** node. The **FaceDefTables** lists all FAPs that animate the face model. The FAPs animate the downloaded face model by updating the 'Transform' or **IndexedFaceSet** nodes of the scenegraph. For each listed FAP, the **FaceDefTables** describes which nodes are animated by this FAP and how they are animated. All FAPs that occur in the bitstream have to be specified in the **FaceDefTables**. The animation generated by a FAP can be specified either by updating a **Transform** node (using a **FaceDefTransform**), or as a deformation of an **IndexedFaceSet** (using a **FaceDefMesh**).

The FAPUs will be calculated by the decoder using the feature points which must be specified in the FDP. The FAPUs are needed in order to animate the downloaded face model.

9.5.3.2.10.3 Detailed Semantics

fapID	specifies the FAP, for which the animation behavior is defined in the **faceDefMesh** and **faceDefTransform** field.
highLevelSelect	specifies the type of viseme or expression, if **fapID** is 1 or 2. In other cases this field has no meaning.
faceDefMesh	contains a **FaceDefMesh** node.
faceDefTransform	contains a **FaceDefTransform** node.

9.5.3.2.11 FaceDefTransform

9.5.3.2.11.1 Semantic Table

```
FaceDefTransform {
    field    SFNode       faceSceneGraphNode    NULL
    field    SFInt32      fieldId               1
    field    SFRotation   rotationDef           0, 0, 1, 0
    field    SFVec3f      scaleDef              1, 1, 1
    field    SFVec3f      translationDef        0, 0, 0
}
```

9.5.3.2.11.2 Main Functionality

Defines which field (**rotation**, **scale** or **translation**) of a **Transform** node **faceSceneGraphNode** of **faceSceneGraph** is updated by a facial animation parameter, and how the field is updated. If the face is in its neutral position, the **faceSceneGraphNode** has its fields **translation**, **scale**, and **rotation** set to the neutral values $(0,0,0)^T$, $(1,1,1)^T$, (0,0,1,0), respectively.

9.5.3.2.11.3 Detailed Semantics

faceSceneGraphNode	**Transform** node for which the animation is defined. The node shall be part of **faceScenegraph** as defined in the FDP node.
fieldId	Specifies which field in the **Transform** node **faceSceneGraphNode** is updated by the FAP during animation. Possible fields are translation, rotation, scale. If**fieldID** ==1, **rotation** will be updated using **rotationDef** and FAPValue. If**fieldID** ==2, **scale** will be updated using **scaleDef** and FAPValue. If**fieldID** ==3, **translation** will be updated using **translationDef** and FAPValue.
rotationDef	is of type **SFRotation**. With **rotationDef**=(r_x,r_y,r_z,θ), the new node value **rotation** of the Transform node **faceSceneGraphNode** is: **rotation**: =$(r_x,r_y,r_z,\theta$*FAPValue*AU) [AU is defined in ISO/IEC FCD 14496-2]
scaleDef	is of type **SFVec3f** The new node value **scale** of the **Transform** node **faceSceneGraphNode** is: **scale**:= FAPValue***scaleDef**
translationDef	is of type **SFVec3f** The new node value **translation** of the **Transform** node **faceSceneGraphNode** is: **translation**:= FAPValue***translationDef**

9.5.3.2.12 FaceDefMesh

9.5.3.2.12.1 Semantic Table

FaceDefMesh {

```
    field     SFNode     faceSceneGraphNode     NULL
    field     MFInt32    intervalBorders        NULL
    field     MFInt32    coordIndex             NULL
    field     MFVec3f    displacements          NULL
}
```

9.5.3.2.12.2 Main Functionality

Defines the piece-wise linear motion trajectories for vertices of the **IndexedFaceSet faceSceneGraphNode** of the **faceSceneGraph** of the **FDP** node, which is deformed by a facial animation parameter.

9.5.3.2.12.3 Detailed Semantics

faceSceneGraphNode The **IndexedFaceSet** node for which the animation is defined. The node shall be part of **faceSceneGraph** as defined in the **FDP** node.

intervalBorders Interval borders for the piece-wise linear approximation in increasing order. Exactly one interval border must have the value 0.

coordIndex A list of indices into the **Coordinate** node of the **IndexedFaceSet** node specified by **faceSceneGraphNode**.

displacements For each vertex indexed in the **coordIndex** field, displacement vectors are given for the intervals defined in the intervalBorders field. There must be exactly (num(IntervalBorders)-1)*num(coordIndex) values in this field.

In most cases, the animation generated by a FAP cannot be specified by updating a **Transform** node, as a deformation of an **IndexedFaceSet** must be performed. In this case, the **FaceDefTables** has to define which **IndexedFaceSets** are affected by a given FAP and how the **coord** fields of these nodes are updated. This is done by means of tables.

If a FAP affects an **IndexedFaceSet**, the **FaceDefTables** has to specify a table of the following format for this **IndexedFaceSet**:

Vertex no.	1st Interval $[I_1, I_2]$	2nd Interval $[I_2, I_3]$	…
Index 1	Displacement D_{11}	Displacement D_{12}	…
Index 2	Displacement D_{21}	Displacement D_{22}	…
…	…	…	…

Exactly one interval border I_k must have the value 0:

$[I_1, I_2], [I_2, I_3], \ldots[I_{k-1}, 0], [0, I_{k+1}], [I_{k+1}, I_{k+2}], \ldots[I_{max-1}, I_{max}]$

During animation, when the decoder receives a FAP, which affects one or more **IndexedFaceSets** of the face model, it piece-wise linearly approximates the motion trajectory of each vertex of the affected **IndexedFaceSets** by using the appropriate table (Figure 9-12).

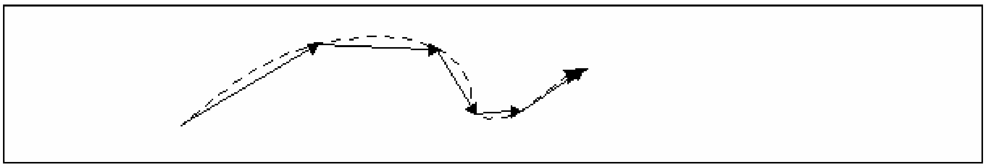


Figure 9-12: An arbitrary motion trajectory is approximated as a piece-wise linear one.

If $\boldsymbol{P}_m$ is the position of the mth vertex in the **IndexedFaceSet** in neutral state (FAP = 0), $\boldsymbol{P'}_m$ the position of the same vertex after animation with the given FAP and $\boldsymbol{D}_{mk}$ the 3D displacement in the k-th interval, the following algorithm shall be applied to determine the new position $\boldsymbol{P'}_m$:

1. Determine, in which of the intervals listed in the table the received FAP is lying.
2. If the received FAP is lying in the jth interval [Ij, Ij+1] and 0=Ik ≤ Ij, the new vertex position P'm of the mth vertex of the IndexedFaceSet is given by:

P’m = FAPU * ((Ik+1-0) * Dm,k + (Ik+2-Ik+1) * Dm, k+1 + …(Ij - Ij-1) * Dm, j-1 + (FAP-Ij) * Dm, j) + Pm.

3. If FAP > Imax, then P’m is calculated by using the equation given in 2. and setting the index j = max.
4. If the received FAP is lying in the jth interval [Ij, Ij+1] and Ij+1 ≤ Ik=0, the new vertex position P’m is given by:

   P’m = FAPU * ((Ij+1 - FAP) * Dm, j + (Ij+2 - Ij+1) * Dm, j+1 + … (Ik-1 - Ik-2) * Dm, k-2 + (0 - Ik-1) * Dm, k-1) + Pm.
5. If FAP < I1, then P’m is calculated by using equation 4. and setting the index j+1 = 1.
6. If for a given FAP and ‘IndexedFaceSet’ the table contains only one interval, the motion is strictly linear:

   P’m = FAPU * FAP * Dm1 + Pm.

Example:

```
FaceDefMesh {
   objectDescriptorID UpperLip
   intervalBorders [ -1000, 0, 500, 1000 ]
   coordIndex [ 50, 51]
   displacements [1 0 0,  0.9 0 0, 1.5 0 4, 0.8 0 0, 0.7 0 0, 2 0 0 ]
}
```

This **FaceDefMesh** defines the animation of the mesh “UpperLip”. For the piecewise-linear motion function three intervals are defined: [-1000, 0], [0, 500] and [500, 1000]. Displacements are given for the vertices with the indices 50 and 51. The displacements for the vertex 50 are: (1 0 0), (0.9 0 0) and (1.5 0 4), the displacements for vertex 51 are (0.8 0 0), (0.7 0 0) and (2 0 0). Given a FAPValue of 600, the resulting displacement for vertex 50 would be:

displacement(vertex 50) = 500*$(0.9\ 0\ 0)^T$ + 100 * $(1.5\ 0\ 4)^T$ = $(600\ 0\ 400)^T$.

If the **FAPValue** is outside the given intervals, the boundary intervals are extended to +I or -I, as appropriate.

9.5.3.3 3D Non-Native Nodes

These nodes have their semantics specified in ISO/IEC 14772-1 with further restrictions and extensions defined herein.

9.5.3.3.1 Background

9.5.3.3.1.1 Semantic Table

Background {

eventIn	SFBool	**set_bind**	NULL
exposedField	MFFloat	**groundAngle**	NULL
exposedField	MFColor	**groundColor**	NULL
exposedField	MFString	**backURL**	NULL
exposedField	MFString	**frontURL**	NULL
exposedField	MFString	**leftURL**	NULL
exposedField	MFString	**rightURL**	NULL
exposedField	MFString	**topURL**	NULL
exposedField	MFFloat	**skyAngle**	NULL
exposedField	MFColor	**skyColor**	0, 0, 0
eventOut	SFBool	**isBound**	

}

9.5.3.3.1.2 Detailed Semantics

The **backUrl**, **frontUrl**, **leftUrl**, **rightUrl**, **topUrl** fields specify the data sources to be used (see 9.2.7.1).

9.5.3.3.2 Billboard

9.5.3.3.2.1 Semantic Table

Billboard {

eventIn	MFNode	**addChildren**	NULL
eventIn	MFNode	**removeChildren**	NULL
exposedField	SFVec3f	**axisOfRotation**	0, 1, 0
exposedField	MFNode	**children**	NULL
field	SFVec3f	**bboxCenter**	0, 0, 0
field	SFVec3f	**bboxSize**	-1, -1, -1

}

9.5.3.3.3 Box

9.5.3.3.3.1 Semantic Table

Box {

field	SFVec3f	**size**	2, 2, 2

}

9.5.3.3.4 Collision

9.5.3.3.4.1 Semantic Table

Collision {

eventIn	MFNode	**addChildren**	NULL
eventIn	MFNode	**removeChildren**	NULL
exposedField	MFNode	**children**	NULL
exposedField	SFBool	**collide**	TRUE
field	SFVec3f	**bboxCenter**	0, 0, 0
field	SFVec3f	**bboxSize**	-1, -1, -1
field	SFNode	**proxy**	NULL
eventOut	SFTime	**collideTime**	

}

9.5.3.3.5 Cone

9.5.3.3.5.1 Semantic Table

Cone {

field	SFFloat	**bottomRadius**	1
field	SFFloat	**height**	2
field	SFBool	**side**	TRUE
field	SFBool	**bottom**	TRUE

}

9.5.3.3.6 Coordinate

9.5.3.3.6.1 Semantic Table

Coordinate {

exposedField	MFVec3f	**point**	NULL

}

9.5.3.3.7 CoordinateInterpolator

9.5.3.3.7.1 Semantic Table

```
CoordinateInterpolator {
    eventIn         SFFloat     set_fraction        NULL
    exposedField    MFFloat     key                 NULL
    exposedField    MFVec3f     keyValue            NULL
    eventOut        MFVec3f     value_changed
}
```

9.5.3.3.8 Cylinder

9.5.3.3.8.1 Semantic Table

```
Cylinder {
    field           SFBool      bottom              TRUE
    field           SFFloat     height              2
    field           SFFloat     radius              1
    field           SFBool      side                TRUE
    field           SFBool      top                 TRUE
}
```

9.5.3.3.9 DirectionalLight

9.5.3.3.9.1 Semantic Table

```
DirectionalLight {
    exposedField    SFFloat     ambientIntensity    0
    exposedField    SFColor     color               1, 1, 1
    exposedField    SFVec3f     direction           0, 0, -1
    exposedField    SFFloat     intensity           1
    exposedField    SFBool      on                  TRUE
}
```

9.5.3.3.10 ElevationGrid

9.5.3.3.10.1 Semantic Table

```
ElevationGrid {
    eventIn         MFFloat     set_height          NULL
    exposedField    SFNode      color               NULL
    exposedField    SFNode      normal              NULL
    exposedField    SFNode      texCoord            NULL
    field           MFFloat     height              NULL
    field           SFBool      ccw                 TRUE
    field           SFBool      colorPerVertex      TRUE
    field           SFFloat     creaseAngle         0
    field           SFBool      normalPerVertex     TRUE
    field           SFBool      solid               TRUE
    field           SFInt32     xDimension          0
    field           SFFloat     xSpacing            1
    field           SFInt32     zDimension          0
    field           SFFloat     zSpacing            1
}
```

9.5.3.3.11 Extrusion

9.5.3.3.11.1 Semantic Table

Extrusion {

eventIn	MFVec2f	**set_crossSection**	NULL
eventIn	MFRotation	**set_orientation**	NULL
eventIn	MFVec2f	**set_scale**	NULL
eventIn	MFVec3f	**set_spine**	NULL
field	SFBool	**beginCap**	TRUE
field	SFBool	**ccw**	TRUE
field	SFBool	**convex**	TRUE
field	SFFloat	**creaseAngle**	0
field	MFVec2f	**crossSection**	1, 1, 1, -1, -1, -1, -1, 1, 1, 1
field	SFBool	**endCap**	TRUE
field	MFRotation	**orientation**	0, 0, 1, 0
field	MFVec2f	**scale**	1, 1
field	SFBool	**solid**	TRUE
field	MFVec3f	**spine**	0, 0, 0, 0, 1, 0

}

9.5.3.3.12 Group

9.5.3.3.12.1 Semantic Table

Group {

EventIn	MFNode	**addChildren**	NULL
EventIn	MFNode	**removeChildren**	NULL
ExposedField	MFNode	**children**	NULL
Field	SFVec3f	**bboxCenter**	0, 0, 0
Field	SFVec3f	**bboxSize**	-1, -1, -1

}

9.5.3.3.12.2 Detailed Semantics

If multiple subgraphs containing audio content (i.e., **Sound** nodes) are children of a **Group**node, the sounds are combined as follows:

If all of the children have equal numbers of channels, or are each a spatially-presented sound, the sound outputs of the children sum to create the audio output of this node.

If the children do not have equal numbers of audio channels, or some children, but not all, are spatially presented sounds, the semantics are TBD.

Kommentar: What's the right way to do this? Liam just deleted "the semantics are TBD", but the remaining text does not make sense.

9.5.3.3.13 IndexedFaceSet

9.5.3.3.13.1 Semantic Table

IndexedFaceSet {

eventIn	MFInt32	**set_colorIndex**	NULL
eventIn	MFInt32	**set_coordIndex**	NULL
eventIn	MFInt32	**set_normalIndex**	NULL
eventIn	MFInt32	**set_texCoordIndex**	NULL
exposedField	SFNode	**color**	NULL
exposedField	SFNode	**coord**	NULL
exposedField	SFNode	**normal**	NULL
exposedField	SFNode	**texCoord**	NULL
field	SFBool	**ccw**	TRUE
field	MFInt32	**colorIndex**	NULL

```
    field          SFBool      colorPerVertex      TRUE
    field          SFBool      convex              TRUE
    field          MFInt32     coordIndex          NULL
    field          SFFloat     creaseAngle         0
    field          MFInt32     normalIndex         NULL
    field          SFBool      normalPerVertex     TRUE
    field          SFBool      solid               TRUE
    field          MFInt32     texCoordIndex       NULL
}
```

9.5.3.3.13.2 Main Functionality

The **IndexedFaceSet** node represents a 3D polygon mesh formed by constructing faces (polygons) from points specified in the **coord** field. If the **coordIndex** field is not NULL, **IndexedFaceSet** uses the indices in its **coordIndex** field to specify the polygonal faces by connecting together points from the **coord** field. An index of -1 shall indicate that the current face has ended and the next one begins. The last face may be followed by a -1. **IndexedFaceSet** shall be specified in the local coordinate system and shall be affected by parent transformations.

9.5.3.3.13.3 Detailed Semantics

The **coord** field specifies the vertices of the face set and is specified by **Coordinate** node.

If the **coordIndex** field is not NULL, the indices of the **coordIndex** field shall be used to specify the faces by connecting together points from the **coord** field. An index of -1 shall indicate that the current face has ended and the next one begins. The last face may followed by a -1.

If the **coordIndex** field is NULL, the vertices of the **coord** field are laid out in their respective order to specify one face.

If the **color** field is NULL and there is a **Material** defined for the **Appearance** affecting this **IndexedFaceSet**, then the **emissiveColor** of the **Material** shall be used to draw the faces.

9.5.3.3.14 IndexedLineSet

9.5.3.3.14.1 Semantic Table

```
IndexedLineSet {
    eventIn        MFInt32     set_colorIndex      NULL
    eventIn        MFInt32     set_coordIndex      NULL
    exposedField   SFNode      color               NULL
    exposedField   SFNode      coord               NULL
    field          MFInt32     colorIndex          NULL
    field          SFBool      colorPerVertex      TRUE
    field          MFInt32     coordIndex          NULL
}
```

9.5.3.3.15 Inline

9.5.3.3.15.1 Semantic Table

```
Inline {
    exposedField   MFString    url                 NULL
    field          SFVec3f     bboxCenter          0, 0, 0
    field          SFVec3f     bboxSize            -1, -1, -1
}
```

9.5.3.3.15.2 Detailed Semantics

The **url** field specifies the data source to be used (see 9.2.7.1). The external source must contain a valid BIFS scene, and may include BIFS-Commands and BIFS-Anim frames

9.5.3.3.16 LOD

9.5.3.3.16.1 Semantic Table

```
LOD {
    exposedField   MFNode     level          NULL
    field          SFVec3f    center         0, 0, 0
    field          MFFloat    range          NULL
    field          MFFloat    fpsRange       NULL
}
```

9.5.3.3.17 Material

9.5.3.3.17.1 Semantic Table

```
Material {
    exposedField   SFFloat    ambientIntensity   0.2
    exposedField   SFColor    diffuseColor       0.8, 0.8, 0.8
    exposedField   SFColor    emissiveColor      0, 0, 0
    exposedField   SFFloat    shininess          0.2
    exposedField   SFColor    specularColor      0, 0, 0
    exposedField   SFFloat    transparency       0
}
```

9.5.3.3.18 Normal

9.5.3.3.18.1 Semantic Table

```
Normal {
    exposedField   MFVec3f    vector         NULL
}
```

9.5.3.3.19 NormalInterpolator

9.5.3.3.19.1 Semantic Table

```
NormalInterpolator {
    eventIn        SFFloat    set_fraction   NULL
    exposedField   MFFloat    key            NULL
    exposedField   MFVec3f    keyValue       NULL
    eventOut       MFVec3f    value_changed
}
```

9.5.3.3.20 OrientationInterpolator

9.5.3.3.20.1 Semantic Table

```
OrientationInterpolator {
    eventIn        SFFloat      set_fraction   NULL
    exposedField   MFFloat      key            NULL
    exposedField   MFRotation   keyValue       NULL
    eventOut       SFRotation   value_changed
}
```

9.5.3.3.21 PointLight

9.5.3.3.21.1 Semantic Table

PointLight {

exposedField	SFFloat	**ambientIntensity**	0
exposedField	SFVec3f	**attenuation**	1, 0, 0
exposedField	SFColor	**color**	1, 1, 1
exposedField	SFFloat	**intensity**	1
exposedField	SFVec3f	**location**	0, 0, 0
exposedField	SFBool	**on**	TRUE
exposedField	SFFloat	**radius**	100

}

9.5.3.3.22 PointSet

9.5.3.3.22.1 Semantic Table

PointSet {

exposedField	SFNode	**color**	NULL
exposedField	SFNode	**coord**	NULL

}

9.5.3.3.23 PositionInterpolator

9.5.3.3.23.1 Semantic Table

PositionInterpolator {

eventIn	SFFloat	**set_fraction**	NULL
exposedField	MFFloat	**key**	NULL
exposedField	MFVec3f	**keyValue**	NULL
eventOut	SFVec3f	**value_changed**	

}

9.5.3.3.24 ProximitySensor

9.5.3.3.24.1 Semantic Table

ProximitySensor {

exposedField	SFVec3f	**center**	0, 0, 0
exposedField	SFVec3f	**size**	0, 0, 0
exposedField	SFBool	**enabled**	TRUE
eventOut	SFBool	**isActive**	
eventOut	SFVec3f	**position_changed**	
eventOut	SFRotation	**orientation_changed**	
eventOut	SFTime	**enterTime**	
eventOut	SFTime	**exitTime**	

}

9.5.3.3.25 Sound

9.5.3.3.25.1 Semantic Table

Sound {

exposedField	SFVec3f	**direction**	0, 0, 1
exposedField	SFFloat	**intensity**	1
exposedField	SFVec3f	**location**	0, 0, 0

```
    exposedField   SFFloat   maxBack      10
    exposedField   SFFloat   maxFront     10
    exposedField   SFFloat   minBack      1
    exposedField   SFFloat   minFront     1
    exposedField   SFFloat   priority     0
    exposedField   SFNode    source       NULL
    field          SFBool    spatialize   TRUE
}
```

9.5.3.3.25.2 Main Functionality

The **Sound** node is used to attach sound to a scene, thereby giving it spatial qualities and relating it to the visual content of the scene.

The **Sound** node relates an audio BIFS sub-tree to the rest of an audiovisual scene. By using this node, sound may be attached to a group, and spatialized or moved around as appropriate for the spatial transforms above the node. By using the functionality of the audio BIFS nodes, sounds in an audio scene dscribed using this Final Committee Draft of International Standard may be filtered and mixed before being spatially composited into the scene.

9.5.3.3.25.3 Detailed Semantics

The semantics of this node are as defined in ISO/IEC 14772-1, Subclause 6.42, with the following exceptions and additions:

The **source** field allows the connection of an audio source containing the sound.

The **spatialize** field determines whether the Sound should be spatialized. If this flag is set, the sound should be presented spatially according to the local coordinate system and current **listeningPoint**, so that it apparently comes from a source located at the **location** point, facing in the direction given by **direction**. The exact manner of spatialization is implementation-dependant, but implementators are encouraged to provide the maximum sophistication possible depending on terminal resources.

If there are multiple channels of sound output from the child Sound, they may or may not be spatialized, according to the **phaseGroup** properties of the child, as follows. Any individual channels, that is, channels not phase-related to other channels, are summed linearly and then spatialized. Any phase-grouped channels are not spatialized, but passed through this node unchanged. The sound presented in the scene is thus a single spatialized sound, represented by the sum of the individual channels, plus an “ambient” sound represented by mapping all the remaining channels into the presentation system as discussed in Subclause 9.2.14.2.2.

If the **spatialize** field is not set, the audio channels from the child are passed through unchanged, and the sound presented in the scene due to this node is an “ambient” sound represented by mapping all the audio channels output by the child into the presentation system as discussion in Subclause 9.2.14.2.2.

9.5.3.3.25.4 Nodes above the Sound node

As with the visual objects in the scene, the **Sound** node may be included as a child or descendant of any of the grouping or transform nodes. For each of these nodes, the sound semantics are as follows:

Affine transformations presented in the grouping and transform nodes affect the apparant spatialization position of spatialized sound. They have no effect on “ambient” sounds.

If a particular grouping or transform node has multiple **Sound** nodes as descendants, then they are combined for presentation as follows. Each of the **Sound** nodes may be producing a spatialized sound, a multichannel ambient sound, or both. For all of the spatialized sounds in descendant nodes, the sounds are linearly combined through simple summation from presentation. For multichannel ambient sounds, the sounds are linearly combined channel-by-channel for presentation.

Example: **Sound** node S1 generates a spatialized sound s1 and five channels of multichannel ambient sound a1[1-5]. **Sound** node S2 generates a spatialized sound s2 and two channels of multichannel ambient sound a2[1-2]. S1 and S2 are grouped under a single **Group**node. The resulting sound is the superposition of the spatialized sound s1, the spatialized sound s2, and the five-channel ambient multichannel sound represented by a3[1-5], where

a3[1] = a1[1] + a2[1]

a3[2] = a1[2] + a2[2]

a3[3] = a1[3]

a3[4] = a1[4]

a3[5] = a1[5].

9.5.3.3.26 Sphere

9.5.3.3.26.1 Semantic Table

Sphere {

field	SFFloat	**radius**	1

}

9.5.3.3.27 SpotLight

9.5.3.3.28 Semantic Table

SpotLight {

exposedField	SFFloat	**ambientIntensity**	0
exposedField	SFVec3f	**attenuation**	1, 0, 0
exposedField	SFFloat	**beamWidth**	1.5708
exposedField	SFColor	**color**	1, 1, 1
exposedField	SFFloat	**cutOffAngle**	0.785398
exposedField	SFVec3f	**direction**	0, 0, -1
exposedField	SFFloat	**intensity**	1
exposedField	SFVec3f	**location**	0, 0, 0
exposedField	SFBool	**on**	TRUE
exposedField	SFFloat	**radius**	100

}

9.5.3.3.29 Transform

9.5.3.3.29.1 Semantic Table

Transform {

eventIn	MFNode	**addChildren**	NULL
eventIn	MFNode	**removeChildren**	NULL
exposedField	SFVec3f	**center**	0, 0, 0
exposedField	MFNode	**children**	NULL
exposedField	SFRotation	**rotation**	0, 0, 1, 0
exposedField	SFVec3f	**scale**	1, 1, 1
exposedField	SFRotation	**scaleOrientation**	0, 0, 1, 0
exposedField	SFVec3f	**translation**	0, 0, 0
field	SFVec3f	**bboxCenter**	0, 0, 0
field	SFVec3f	**bboxSize**	-1, -1, -1

}

9.5.3.3.29.2 Detailed Semantics

If some of the child subgraphs contain audio content (i.e., the subgraphs contain **Sound** nodes), the child sounds are transformed and mixed as follows.

If each of the child sounds is a spatially presented sound, the **Transform** node applies to the local coordinate system of the **Sound** nodes to alter the apparent spatial location and direction. The spatialized outputs of the children nodes sum equally to produce the output at this node.

If the children are not spatially presented but have equal numbers of channels, the **Transform** node has no effect on the childrens' sounds. The child sounds are summed equally to produce the audio output at this node.

If some children are spatially-presented and some not, or all children do not have equal numbers of channels, the semantics are not defined.

9.5.3.3.30 Viewpoint

9.5.3.3.30.1 Semantic Table

Viewpoint {

eventIn	SFBool	**set_bind**	NULL
exposedField	SFFloat	**fieldOfView**	0.785398
exposedField	SFBool	**jump**	TRUE
exposedField	SFRotation	**orientation**	0, 0, 1, 0
exposedField	SFVec3f	**position**	0, 0, 10
field	SFString	**description**	""
eventOut	SFTime	**bindTime**	
eventOut	SFBool	**isBound**	

}

9.5.4 Mixed 2D/3D Nodes

9.5.4.1 Mixed 2D/3D Nodes Overview

Mixed 2D and 3D nodes enable to build scenes made up of 2D and 3D primitives together. In particular, it is possible to view simultaneously several rendered 2D and 3D scenes, to use a rendered 2D or 3D scene as a texture map, or to render a 2D scene in a 3D local coordinate plane.

9.5.4.2 2D/3D Native Nodes

9.5.4.2.1 Layer2D

9.5.4.2.1.1 Semantic Table

Layer2D {

eventIn	MFNode	**addChildren**	NULL
eventIn	MFNode	**removeChildren**	NULL
eventIn	MFNode	**addChildrenLayer**	NULL
eventIn	MFNode	**removeChildrenLayer**	NULL
exposedField	MFNode	**children**	NULL
exposedField	MFNode	**childrenLayer**	NULL
exposedField	SFVec2f	**size**	-1, -1
exposedField	SFVec2f	**translation**	0, 0
exposedField	SFInt32	**depth**	0
field	SFVec2f	**bboxCenter**	0, 0
field	SFVec2f	**bboxSize**	-1, -1

}

9.5.4.2.1.2 Main Functionality

The **Layer2D** node is a transparent rendering rectangle region on the screen where a 2D scene is shown. The **Layer2D** is part of the layers hierarchy, and can be composed in a 2D environment with depth.

Layer 2D and **Layer3D** nodes enable the composition in a 2D space with depth of multiple 2D and 3D scenes. This allows users, for instance, to have 2D interfaces to a 2D scene; or 3D interfaces to a 2D scene, or to view a 3D scene from different view points in the same scene. Interaction with objects drawn inside a **Layer** node is enabled only on the top most layer of the scene at a given position of the rendering area. This means that it is impossible to interact with an object behind another layer.

9.5.4.2.1.3 Detailed Semantics

The **addChildren** eventIn specifies a list of 2D nodes that must be added to the **Layer2D's children** field.

The **removeChildren** eventIn specifies a list of 2D nodes that must be removed from the **Layer2D's children** field.

The **addChildrenLayer** eventIn specifies a list of 2D nodes that must be added to the **Layer2D's childrenLayer** field.

The **removeChildrenLayer** eventIn specifies a list of 2D nodes that must be removed from the **Layer2D's childrenLayer** field.

The **children** field may contain any 2D children nodes that define a 2D scene. For any 2D grouping node, the order of children is the order of drawing for 2D primitives.

The **childrenLayer** field can specify either a 2D or 3D layer node.

The layering of the 2D and 3D layers is specified by the **translation** and **depth** fields. The units of 2D scenes are used for the translation parameter. The **size** parameter is given as a floating point number which expresses a fraction of the width and height of the parent layer. In case of a layer at the root of the hierarchy, the fraction is a fraction of the screen rendering area. A size of -1 in one direction, means that the **Layer2D** node is not specified in size in that direction, and that the size is adjusted to the size of the parent layer, or the global rendering area dimension if the layer is on the top of the hierarchy.

In the case where a 2D scene or object is shared between several **Layer2D**, the behaviours are defined exactly as for objects which are multiply referenced using the DEF/USE mechanism. A sensor triggers an event whenever the sensor is triggered in any of the **Layer2D** in which it is contained. The behaviors triggered by the shared sensors as well as other behaviors that apply on objects shared between several layers apply on all layers containing these objects.

All the 2D objects contained in a single **Layer2D** node form a single composed object. This composed object is viewed by other objects as a single object. In other words, if a **Layer2D** node A is the parent of two objects B and C layered one on top of the other, it will not be possible to insert a new object D between B and C unless D is added as a child of A.

The **bboxCenter** field specifies the center of the bounding box and the **bboxSize** field specifies the width and the height of the bounding box. It is possible not to transmit the **bboxCenter** and **bboxSize** fields, but if they are transmitted, the corresponding box must contain all the children. The behaviour of the terminal in other cases is not specified. The bounding box semantic is described in more detail in Subclause 9.2.13.

9.5.4.2.2 Layer3D

9.5.4.2.2.1 Semantic Table

```
Layer3D {
    eventIn        MFNode    addChildren            NULL
    eventIn        MFNode    removeChildren         NULL
    eventIn        MFNode    addChildrenLayer       NULL
    eventIn        MFNode    removeChildrenLayer    NULL
    exposedField   MFNode    children               NULL
    exposedField   MFNode    childrenLayer          NULL
    exposedField   SFVec2f   translation            0, 0
    exposedField   SFInt32   depth                  0
    exposedField   SFVec2f   size                   -1, -1
    eventIn        SFNode    background             NULL
    eventIn        SFNode    fog                    NULL
    eventIn        SFNode    navigationInfo         NULL
    eventIn        SFNode    viewpoint              NULL
    Field          SFVec3f   bboxCenter             0, 0, 0
    Field          SFVec3f   bboxSize               -1, -1, -1
}
```

9.5.4.2.2.2 Main Functionality

The **Layer3D** node is a transparent rendering rectangle region on the screen where a 3D scene is shown. The **Layer3D** is part of the layers hierarchy, and can be composed in a 2D environment with depth

9.5.4.2.2.3 Detailed Semantics

Layer3D is composed as described in the **Layer2D** specification. For navigation in a **Layer3D**, the same principle as for interaction applies. It is possible to navigate any **Layer3D** node that appears as the front layer at a given position on the rendering area. A terminal must provide a way to select an active layer among all the displayed **Layer3D**. Once this layer is selected, the navigation acts on this layer. For instance; if a mouse pointing device is used, the active layer for navigation may be the front layer under the starting position of the mouse dragging action for navigating the 3D scene.

The **addChildren** eventIn specifies a list of 3D nodes that must be added to the **Layer3D's children** field.

The **removeChildren** eventIn specifies a list of 3D nodes that must be removed from the **Layer3D's children** field.

The **addChildrenLayer** eventIn specifies a list of 3D nodes that must be added to the **Layer3D's childrenLayer** field.

The **removeChildrenLayer** eventIn specifies a list of 3D nodes that must be removed from the **Layer3D's childrenLayer** field.

The **children** field may specify any 3D children nodes that define a 3D scene.

The **childrenLayer** field may specify either a 2D or 3D layer. The layering of the 2D and 3D layers is specified by the **translation** and **depth** fields. The **translation** field is expressed, as in the case of the Layer2D in terms of 2D units.

The **size** parameter has the same semantic and units as in the **Layer2D**.

The **bboxCenter** field specifies the center of the bounding box and the **bboxSize** field specifies the width and the height of the bounding box. It is possible not to transmit the **bboxCenter** and **bboxSize** fields, but if they are transmitted, the corresponding box must contain all the children. The behaviour of the terminal in other cases is not specified. The bounding box semantic is described in more detail in Subclause 9.2.13.

A **Layer3D** stores the stack of bindable leaf nodes of the children scene of the layer. All bindable leaf nodes are eventIn fields of the **Layer3D** node. At run-time, these fields take the value of the currently bound bindable leaf nodes for the 3D scene that is a child of the **Layer3D** node. This will allow, for instance, to set a current viewpoint to a **Layer3D**, in response to some event. Note that this cannot be achieved by a direct use of the **set_bind** eventIn of the **Viewpoint** nodes since scenes or nodes can be shared between different layers. If a **set_bind** TRUE event is sent to the **set_bind** eventIn of any of the bindable leaf nodes, then all **Layer3D** nodes having this node as a child node will set this node as the current bindable leaf node.

In the case where a 3D scene or object is shared between several **Layer3D**, the behaviours are defined exactly as for objects which are multiply referenced using the DEF/USE mechanism. A sensor triggers an event whenever the sensor is triggered in any of the **Layer3D** in which it is contained. The behaviors triggered by the shared sensors as well as other behaviors that apply on objects shared between several layers apply on all layers containing these objects.

All the 3D objects under a same **Layer3D** node form a single composed object. This composed object is viewed by other objects as a single object.

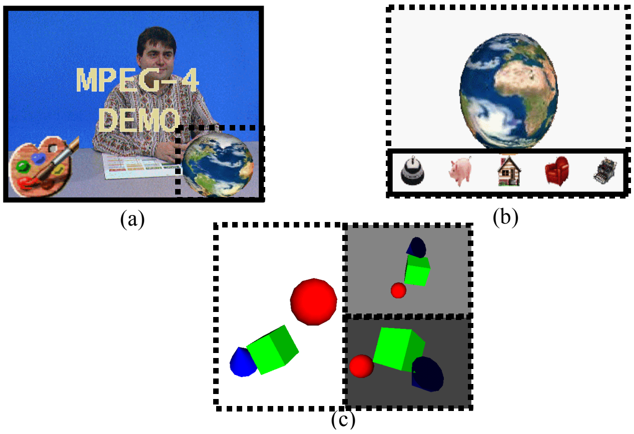


Figure 9-13: Three Layer2D and Layer3D examples. Layer2D are signaled by a plain line, Layer3D with a dashed line. Image (a) shows a Layer3D containing a 3D view of the earth on top of a Layer2D composed of a video, a logo and a text. Image (b) shows a Layer3D of the earth with a Layer2D containing various icons on top. Image (c) shows 3 views of a 3D scene with 3 non overlaping Layer3D.

9.5.4.2.3 Composite2DTexture

9.5.4.2.3.1 Semantic Table

Composite2DTexture {

exposedField	MFNode	**children**	NULL
exposedField	SFInt32	**pixelWidth**	-1
exposedField	SFInt32	**pixelHeight**	-1

}

9.5.4.2.3.2 Main Functionality

The **Composite2DTexture** node represents a texture that is composed of a 2D scene, which may be mapped onto a 2D object.

9.5.4.2.3.3 Detailed Semantics

All behaviors and user interaction are enabled when using a **Composite2DTexture**. However, sensors contained in the scene which forms the **Composite2DTexture** shall be ignored.

The **children2D** field contains a list of 2D children nodes that define the 2D scene that is to form the texture map. As for any 2D grouping node, the order of children is the order of drawing for 2D primitives.

The **pixelWidth** and **pixelHeight** fields specifies the ideal size in pixels of this map. If left as default value, an undefined size will be used. This is a hint for the content creator to define the quality of the texture mapping.

Figure 9-14: A Composite2DTexture example. The 2D scene is projected on the 3D cube

9.5.4.2.4 Composite3DTexture

9.5.4.2.4.1 Semantic Table

Composite3DTexture {

exposedField	MFNode	**children**	NULL
exposedField	SFInt32	**pixelWidth**	-1
exposedField	SFInt32	**pixelHeight**	-1
eventIn	SFNode	**background**	NULL
eventIn	SFNode	**fog**	NULL
eventIn	SFNode	**navigationInfo**	NULL
eventIn	SFNode	**Viewpoint**	NULL

}

9.5.4.2.4.2 Main Functionality

The **Composite3DTexture** node represents a texture mapped onto a 3D object which is composed of a 3D scene.

9.5.4.2.4.3 Detailed Semantics

Behaviors and user interaction are enabled when using a Composite3DTexture. However, no user navigation is possible on the textured scene and sensors contained in the scene which forms the **Composite3DTexture** shall be ignored.

The **children** field is the list of 3D root and children nodes that define the 3D scene that forms the texture map.

The **pixelWidth** and **pixelHeight** fields specifies the ideal size in pixels of this map. If no value is specified, an undefined size will be used. This is a hint for the content creator to define the quality of the texture mapping.

The **background**, **fog**, **navigationInfo** and **viewpoint** fields represent the current values of the bindable leaf nodes used in the 3D scene. The semantic is the same as in the case of the **Layer3D** node. This node can only be used as a **texture** field of an **Appearence** node.

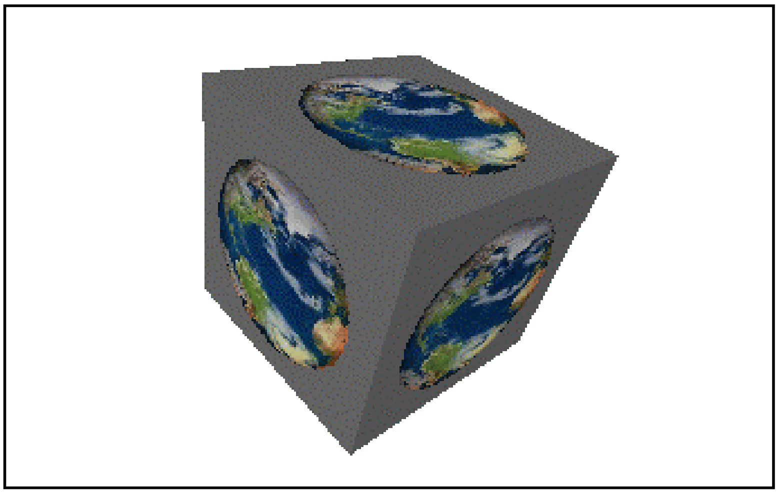


Figure 9-15: A Composite3Dtexture example: The 3D view of the earth is projected onto the 3D cube

9.5.4.2.5 CompositeMap

9.5.4.2.5.1 Semantic Table

```
CompositeMap {
    exposedField    MFNode     children2D      NULL
    exposedField    SFVec2f    sceneSize       -1, -1
}
```

9.5.4.2.5.2 Main Functionality

The **CompositeMap** node allows a 2D scene to appear on a plane in a 3D scene. A similar functionality can be achieved with the **Composite2DTexture**, but when using the **CompositeMap** texture mapping is not necessary to draw the 2D scene in the local XY plane.

9.5.4.2.5.3 Detailed Semantics

When using a **CompositeMap**, the behaviors of 2D objects are similar to those of the 2D scene as drawn in a 2D layer.

The **children** field is the list of 2D root and children nodes that define the 2D scene to be rendered in the local XY coordinate plane. As for any 2D grouping node, the order of children is the order of drawing for 2D primitives.

The **sceneSize** field specifies the size in the local 3D coordinate system of the rendering area where the 2D composited scene needs to be rendered. If no value is specified, the scene rendering area is defined as the rectangle which diagonal is delimited by the origin of the local coordinate system and point (1,1,0) in the local coordinate system.

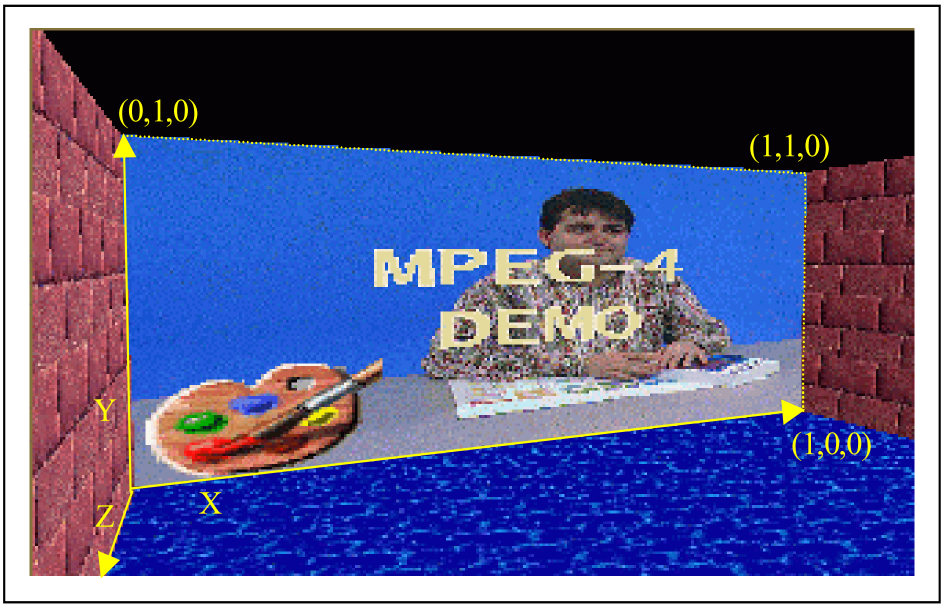


Figure 9-16: A CompositeMap example: The 2D scene as defined in Fig. yyy composed of an image, a logo, and a text, is drawn in the local X,Y plane of the back wall.

10. Synchronization of Elementary Streams

10.1 Introduction

This clause 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 Subclause 7.3. The syntax and semantics to convey these elements to a receiving terminal are embodied in the sync layer, specified in Subclause 10.2.2. This syntax is configurable to adapt to the needs of different types of elementary streams. The required configuration information is specified in Subclause 10.2.3.

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 is best described by a conceptual interface between both layers, termed the Elementary Stream Interface (ESI). The ESI is a Reference Point that need not be accessible in an implementation. It is described in Subclause 10.3.

SL-packetized streams are made available to a delivery mechanism that is outside the scope of this specification. This delivery mechanism is only described in terms of a conceptual Stream Multiplex Interface (SMI) that specifies the information that needs to be exchanged between the sync layer and the delivery mechanism. The SMI is a Reference Point that need not be accessible in an implementation. The SMI may be embodied by the DMIF Application Interface specified in Part 6 of this Final Committee Draft of International Standard. The required properties of the SMI are described in Subclause 10.4.

Note: The delivery mechanism described by the SMI serves to abstract transmission as well as storage. The basic data transport feature that this delivery mechanism shall provide is the framing of the data packets generated by the sync layer. The FlexMux tool (see Subclause 11.2) is an example for such a tool that may be used in the delivery protocol stack if desired..

The items specified in this subclause are depicted in Figure 10-1 below.

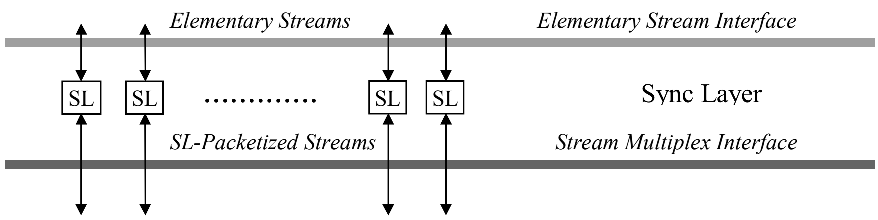


Figure 10-1: Layered ISO/IEC 14496 System

10.2 Sync Layer

10.2.1 Overview

The sync layer (SL) specifies a syntax for the packetization of elementary streams into access units or parts thereof. 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 and their content is opaque. They 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 Subclause 7.3 that defines the timing aspects of the Systems Decoder Model. The SL packet header is configurable as specified in Subclause 10.2.3. The SL packet header itself is specified in Subclause 10.2.4.

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 FlexMux tool specified in Subclause 11.2. 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 Subclause 8.3.3) in the SL packet header. This association must be conveyed through a stream map table using the appropriate signalling means of the delivery mechanism.

10.2.2 SL Packet Specification

10.2.2.1 Syntax

```
class SL_Packet (SLConfigDescriptor SL) {
   aligned(8) SL_PacketHeader slPacketHeader(SL);
   aligned(8) SL_PacketPayload slPacketPayload;
}
```

10.2.2.2 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 Subclause 10.2.4.

`slPacketPayload` – an `SL_PacketPayload` that contains an opaque payload.

10.2.3 SL Packet Header Configuration

10.2.3.1 Syntax

```
aligned(8) class SLConfigDescriptor : bit(8) tag=SLConfigDescrTag {
   bit(8) length;
   bit(8) predefined;
   if (predefined==0) {
      bit(1) useAccessUnitStartFlag;
      bit(1) useAccessUnitEndFlag;
      bit(1) useRandomAccessPointFlag;
      bit(1) usePaddingFlag;
      bit(1) useTimeStampsFlag;
      bit(1) useWallClockTimeStampFlag;
      bit(1) useIdleFlag;
      bit(1) durationFlag;
      bit(32) timeStampResolution;
      bit(32) OCRResolution;
      bit(8) timeStampLength; // must be less than 64
      bit(8) OCRLength;    // must be less than 64
      bit(8) AU_Length;    // must be less than 32
      bit(8) instantBitrateLength;
      bit(4) degradationPriorityLength;
      bit(4) seqNumLength;
      if (durationFlag) {
         bit(32) timeScale;
         bit(16) accessUnitDuration;
         bit(16) compositionUnitDuration;
      }
      if (!useTimeStampsFlag) {
         if (useWallClockTimeStampFlag)
               double(64) wallClockTimeStamp;
         bit(timeStampLength) startDecodingTimeStamp;
         bit(timeStampLength) startCompositionTimeStamp;
      }
   }
```

```
   aligned(8) bit(1) OCRstreamFlag;
   const bit(7) reserved=0b1111.111;
   if (OCRstreamFlag)
      bit(16) OCR_ES_Id;
}
```

10.2.3.2 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).

`length` – length of the remainder of this descriptor in bytes.

`predefined` – allows to default the values from a set of predefined parameter sets as detailed below.

Table 10-1: Overview of predefined `SLConfigDescriptor` values

`predefined` field value	**Description**
0x00	custom
0x01	null SL packet header
0x02 - 0xFF	Reserved for ISO use

Table 10-2: Detailed predefined SLConfigDescriptor values

predefined field value	0x01
`useAccessUnitStartFlag`	0
`useAccessUnitEndFlag`	0
`useRandomAccessPointFlag`	0
`usePaddingFlag`	0
`useTimeStampsFlag`	0
`useWalClockTimeStampFlag`	-
`useIdleFlag`	0
`durationFlag`	-
`timeStampResolution`	-
`OCRResolution`	-
`timeStampLength`	-
`OCRlength`	-
`AU_length`	0
`instantBitrateLength`	-
`degradationPriorityLength`	0
`seqNumLength`	0
`timeScale`	-
`accessUnitDuration`	-
`compositionUnitDuration`	-
`wallClockTimeStamp`	-
`startDecodingTimeStamp`	-
`startCompositionTimeStamp`	-

Kommentar: This needs to be cross-checked.

`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.

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 accessUnitRate, compositionUnitRate, startDecodingTimeStamp and startCompositionTimeStamp conveyed in this SL packet header configuration shall be used for synchronisation.

useWallClockTimeStampFlag – indicates that wallClockTimeStamps are present in this elementary stream. If useTimeStampsFlag equals to zero, only one such time stamp is present in this SLConfigDescriptor.

useIdleFlag – indicates that idleFlag is used in this elementary stream.

durationFlag – indicates that the constant duration of access units and composition units 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 one and 63 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 one and 63 bit.

AU_Length – is the length of the accessUnitLength fields in SL packet headers for this elementary stream. AU_Length shall take values between one and 31 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.

seqNumLength – is the length of the sequenceNumber 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.

wallClockTimeStamp – is a wall clock time stamp in SFTime format that indicates the current wall clock time corresponding to the time indicated by startCompositionTimeStamp.

Kommentar: This cannot be SFTime without specifying the epoch. Also, floats/doubles are not a good idea for time stamps. I think the OCI date stuff should be used instead.

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.

OCRstreamFlag – indicates that an OCR_ES_ID syntax element will follow.

OCR_ES_ID – indicates the elementary stream from which the time base for this elementary stream is derived. This OCR_ES_Id must be unique within its name scope.

10.2.4 SL Packet Header Specification

10.2.4.1 Syntax

```
aligned(8) class SL_PacketHeader (SLConfigDescriptor SL) {
   if (SL.useAccessUnitStartFlag)
      bit(1) accessUnitStartFlag;
   else
      bit(0) accessUnitStartFlag = // see semantics below
   if (SL.useRandomAccessPointFlag)
      bit(1) randomAccessPointFlag;
   if (SL.useAccessUnitEndFlag)
      bit(1) accessUnitEndFlag;
   else
      bit(0) accessUnitEndFlag = // see semantics below
   if (SL.OCRLength>0)
      bit(1) OCRflag;
   if (SL.useIdleFlag)
      bit(1) idleFlag;
   if (SL.usePadding)
      bit(1) paddingFlag;
   if (paddingFlag)
      bit(3) paddingBits;

   if (!idleFlag && (!paddingFlag || paddingBits!=0)) {
      if (SL.seqNumLength>0)
         bit(SL.seqNumLength) sequenceNumber;
      if (OCRflag)
         bit(SL.OCRlength) objectClockReference;

      if (accessUnitStartFlag) {
         if (SL.useTimeStampsFlag) {
            bit(1) decodingTimeStampFlag;
            bit(1) compositionTimeStampFlag;
            if (SL.useWallClockTimeStampFlag)
               bit(1) wallClockTimeStampFlag;
         }
         if (SL.instantBitrateLength>0)
            bit(1) instantBitrateFlag;
         if (decodingTimeStampFlag)
            bit(SL.timeStampLength) decodingTimeStamp;
         if (compositionTimeStampFlag)
            bit(SL.timeStampLength) compositionTimeStamp;
         if (wallClockTimeStampFlag)
            float(64) wallClockTimeStamp;
         if (SL.AU_Length > 0)
            bit(SL.AU_Length)       accessUnitLength;
         if (instantBitrateFlag)
            bit(SL.instantBitrateLength)  instantBitrate;
         if (SL.degradationPriorityLength>0)
            bit(SL.degradationPriorityLength) degradationPriority;
      }
   }
}
```

10.2.4.2 Semantics

`accessUnitStartFlag` – when set to one indicates that an access unit starts in this SL packet. 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 an access unit ends in this SL packet. 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.

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 zero, i. e., no random access points are indicated.

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.

sequenceNumber – 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. This is restricted to a single duplicate per packet. The duplicated SL packet shall immediately follow the original. The sequenceNumber of both 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.

Kommentar: Why only a single duplicate? It costs nothing, implementation-wise, to have more.

objectClockReference – contains an Object Clock Reference time stamp. The OTB time value t is formatted as configured in the associated SLConfigDescriptor, according to the formula:

$$\text{objectClockReference} = \mathrm{NINT}(\text{SL.OCRResolution} * t) \,\%\, 2^{\text{SL.OCRLength}}$$

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:

decodingTimeStampFlag – indicates that a Decoding Time stamp is present in this packet.

compositionTimeStampFlag – indicates that a Composition Time stamp is present in this packet.

wallClockTimeStampFlag – indicates that a wallClockTimeStamp 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 corresponding to the decoding time td of this access unit as configured in the associated SLConfigDescriptor, according to the formula:

decodingTimeStamp = NINT(SL.timeStampResolution * td) % $2^{SL.timeStampLength}$

compositionTimeStamp – is a Composition Time stamp corresponding to the composition time tc of the first composition unit resulting from this access unit as configured in the associated SLConfigDescriptor, according to the formula:

compositionTimeStamp = NINT(SL.timeStampResolution * tc) % $2^{SL.timeStampLength}$

wallClockTimeStamp – is a wall clock Time stamp in SFTime format.

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 of this elementary stream until the next instantBitrate field is found.

degradationPriority – indicates the importance of the payload of this access unit. The streamPriority defines the base priority of an ES. degradationPriority defines a decrease in priority for this access unit relative to the base priority. The priority for this access unit is given by:

AccessUnitPriority = streamPriority - degradationPriority

degradationPriority remains at this value until its next occurrence. This indication is used for graceful degradation by the decoder of this elementary stream. The relative amount of complexity degradation among access units of different elementary streams increases as AccessUnitPriority decreases.

10.2.5 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.

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.

The following indicates recommended values for the SLConfigDescriptor of a Clock Reference Stream:

Table 10-3: SLConfigDescriptor parameter values for a ClockReferenceStream

useAccessUnitStartFlag	0
useAccessUnitEndFlag	0
useRandomAccessPointFlag	0
usePaddingFlag	0
useTimeStampsFlag	0
timeStampResolution	0
timeStampLength	0
useWallClockTimeStampFlag	0

10.3 Elementary Stream Interface (Informative)

The Elementary Stream Interface (ESI) is a conceptual interface that specifies which data need to be exchanged between the entity that generates an elementary stream and the sync layer. Communication between the coding and sync layers cannot only include compressed media, but requires additional information such as time codes, length of access units, etc.

An implementation of this specification, however, does not have to implement the Elementary Stream Interface. It is possible to integrate parsing of the SL-packetized stream and media data decompression in one decoder entity. Note that even in this case the decoder receives a sequence of packets at its input through the Stream Multiplex Interface (see Subclause 10.4) rather than a data stream.

The interface to receive elementary stream data from the sync layer has a number of parameters that reflect the side information that has been retrieved while parsing the incoming SL-packetized stream:

ESI.receiveData (*ESdata, dataLength, decodingTimeStamp, compositionTimeStamp, accessUnitStartFlag, randomAccessFlag, accessUnitEndFlag, degradationPriority, errorStatus*)

ESdata - a number of *dataLength* data bytes for this elementary stream

dataLength - the length in byte of *ESdata*

decodingTimeStamp - the decoding time for the access unit to which this *ESdata* belongs

compositionTimeStamp - the composition time for the access unit to which this *ESdata* belongs

accessUnitStartFlag - indicates that the first byte of *ESdata* is the start of an access unit

randomAccessFlag - indicates that the first byte of *ESdata* is the start of an access unit allowing for random access

accessUnitEndFlag - indicates that the last byte of *ESdata* is the end of an access unit

degradationPriority - indicates the degradation priority for this access unit

errorStatus - indicates whether *ESdata* is error free, possibly erroneous or whether data has been lost preceding the current *ESdata* bytes

A similar interface to send elementary stream data to the sync layer requires the following parameters that will subsequently be encoded on the sync layer:

ESI.sendData (*ESdata, dataLength, decodingTimeStamp, compositionTimeStamp, accessUnitStartFlag, randomAccessFlag, accessUnitEndFlag, degradationPriority*)

ESdata - a number of *dataLength* data bytes for this elementary stream

dataLength - the length in byte of *ESdata*

decodingTimeStamp - the decoding time for the access unit to which this *ESdata* belongs

compositionTimeStamp - the composition time for the access unit to which this *ESdata* belongs

accessUnitStartFlag - indicates that the first byte of *ESdata* is the start of an access unit

randomAccessFlag - indicates that the first byte of *ESdata* is the start of an access unit allowing for random access

accessUnitEndFlag - indicates that the last byte of *ESdata* is the end of an access unit

degradationPriority - indicates the degradation priority for this access unit

10.4 Stream Multiplex Interface (Informative)

The Stream Multiplex Interface (SMI) 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 cannot include only SL-packetized data, but also additional information to convey the length of each SL packet.

An implementation of this specification does not have to expose the Stream Multiplex Interface. A terminal compliant with this Final Committee Draft of International Standard, however, shall have the functionality described by the SMI 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.

A superset of the required SMI functionality is embodied by the DMIF Application Interface specified in Part 6 of this Final Committee Draft of International Standard. 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.

11. Multiplexing of Elementary Streams

11.1 Introduction

Elementary stream data encapsulated in SL-packetized streams are sent/received through the Stream Multiplex Interface, as specified in Subclause 10.4. Multiplexing procedures and the architecture of the delivery protocol layers are outside the scope of this specification. 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 term "TransMux" is used to refer in a generic way to any such protocol stack. Some examples for the embedding of SL-packetized streams in various TransMux protocol stacks are given in Annex C.

The analysis of existing TransMux 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.

11.2 FlexMux Tool

11.2.1 Overview

The FlexMux tool is a flexible multiplexer that accommodates interleaving of SL-packetized streams with varying instantaneous bit rate. The basic data entity of the FlexMux is a FlexMux packet, which has a variable length. One or more SL packets are embedded in a FlexMux packet as specified in detail in the remainder of this subclause. The FlexMux tool provides identification of SL packets originating from different elementary streams by means of FlexMux Channel numbers. Each SL-packetized stream is mapped into one FlexMux Channel. FlexMux packets with data from different SL-packetized streams can therefore be arbitrarily interleaved. The sequence of FlexMux packets that are interleaved into one stream are called a FlexMux Stream.

A FlexMux Stream retrieved from storage or transmission can be parsed as a single data stream without the need for any side information. However, the FlexMux requires framing of FlexMux packets by the underlying layer for random access or error recovery. There is no requirement to frame each individual FlexMux packet. The FlexMux 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 FlexMux.

Two different modes of operation of the FlexMux providing different features and complexity are defined. They are called Simple Mode and MuxCode Mode. A FlexMux Stream may contain an arbitrary mixture of FlexMux packets using either Simple Mode or MuxCode Mode. The syntax and semantics of both modes are specified below.

11.2.2 Simple Mode

In the simple mode one SL packet is encapsulated in one FlexMux packet and tagged by an `index` which is equal to the FlexMux Channel number as indicated in Figure 11-1. This mode does not require any configuration or maintenance of state by the receiving terminal.

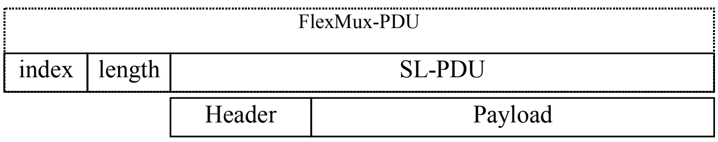


Figure 11-1 : Structure of FlexMux packet in simple mode

11.2.3 MuxCode mode

In the MuxCode mode one or more SL packets are encapsulated in one FlexMux packet as indicated in Figure 11-2. This mode requires configuration and maintenance of state by the receiving terminal. The configuration describes how FlexMux 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 FlexMux packet payload to different FlexMux Channels.

FlexMux-PDU						
index	length	version	SL-PDU	SL-PDU	....	SL-PDU
			H Payld	H Payload	....	H Payload

Figure 11-2: Structure of FlexMux packet in MuxCode mode

11.2.4 FlexMux packet specification

11.2.4.1 Syntax

```
class FlexMuxPacket {
   unsigned int(8) index;
   bit(8) length;
   if (index>239) {
      bit(4) version;
      const bit(4) reserved=0b1111;
      multiple_SL_Packet mPayload;
   } else {
      SL_Packet sPayload;
   }
}
```

Kommentar: I thought that the FlexMux PDU length would be increased!

11.2.4.2 Semantics

The two modes of the FlexMux, Simple Mode and MuxCode Mode are distinguished by the value of `index` as specified below.

`index` – if `index` is smaller than 240 then

FlexMux Channel = `index`

This range of values corresponds to the Simple Mode. If `index` has a value in the range 240 to 255 (inclusive), then the MuxCode Mode is used and a `MuxCode` is referenced as

`MuxCode` = `index` - 240

`MuxCode` is used to associate the `payload` to FlexMux Channels as described in Subclause 11.2.4.3.

Note *Although the number of FlexMux Channels is limited to 256, the use of multiple FlexMux streams allows virtually any number of elementary streams to be provided to the terminal.*

`length` – the length of the FlexMux 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.

`version` – indicates the current version of the `MuxCodeTableEntry` referenced by `MuxCode`. `Version` is used for error resilience purposes. If this version does not match the version of the referenced `MuxCodeTableEntry` that has most recently been received, the FlexMux packet cannot be parsed. The implementation is free to either wait until the required version of `MuxCodeTableEntry` becomes available or to discard the FlexMux packet.

`sPayload` – a single SL packet (Simpe Mode)

`mPayload` – one or more SL packets (MuxCode Mode)

11.2.4.3 Configuration for MuxCode Mode

11.2.4.3.1 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) flexMuxChannel[[i]][[k]];
         bit(8) numberOfBytes[[i]][[k]];
      }
   }
}
```

11.2.4.3.2 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 FlexMux 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 FlexMux 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 FlexMux 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.

`flexMuxChannel[i][k]` – the FlexMux Channel to which the data in this slot belongs.

`numberOfBytes[i][k]` – the number of data bytes in this slot associated to `flexMuxChannel[i][k]`. This number of bytes corresponds to one SL packet.

11.2.5 Usage of MuxCode Mode

The `MuxCodeTableEntry` describes how a FlexMux packet is partitioned into slots that carry data from different FlexMux Channels. This is used as a template for parsing FlexMux packets. If a FlexMux packet is longer than the template, parsing shall resume from the beginning of the template. If a FlexMux packet is shorter than the template, the remainder of the template is ignored.

11.2.5.1 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* (`flexMuxChannel`) with a number of bytes Bytes*n* (`numberOfBytes`). This configuration would result in a splitting of the FlexMux 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 FlexMux packet would be as shown in Figure 11-3.

FlexMux-PDU																
Index	length	version	FMC1	FMC2	FMC1	FMC2	FMC1	FMC2	FMC3	FMC4	FMC5	FMC3	FMC4	FMC5	FMC6	FMC7

Figure 11-3 Example for a FlexMux packet in MuxCode mode

12. Syntactic Description Language

12.1 Introduction

This section describes the mechanism with which bitstream syntax is documented in this Final Committee Draft of International Standard. 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 International Standards ISO/IEC 11172 and 13818 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.

We first describe elementary constructs, then composite syntactic constructs, arithmetic and logical expressions, and finally address syntactic control flow and built-in functions. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure.

12.2 Elementary Data Types

SDL uses the following elementary syntactic elements:

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 decoder.
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 are described in more detail below. Note that all quantities are represented with the most significant byte first, and also with the most significant bit first.

12.2.1 Constant-Length Direct Representation Bit Fields

Constant-length direct representation bit fields are represented as:

Rule E.1: Elementary Data Types
[aligned] *type*[(*length*)] *element_name* [= *value*]; // C++-style comments allowed

The *type* can be any of the following: '*int*' for signed integer, '*unsigned int'* for unsigned integer, '*double*' for floating point, and '*bit*' for raw binary data. '*length'* indicates the length of the element in bits, as it is stored in the bitstream. Note that 'double' can only use 32 or 64 bit lengths. The *value* attribute is only present 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* are always present, except if the data is non-parsable, i.e., it is not included in the bitstream. The attribute 'aligned' means 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)`, can 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 should be interpreted as a 5-bit unsigned integer. By default, data is represented with the most significant bit first, and the most significant byte first.

The value of parsable variables with declaration that fall outside the flow of declarations (see Subclause 12.6, Syntactic Flow Control) is set to 0.

Constants are defined using the 'const' attribute:

```
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 is used, similar to the '0x' prefix for hexadecimal numbers, and a period ('.') can be optionally placed every four digits for readability. Hence 0x0F is equivalent to 0b0000.11111.

In several instances it is desirable to examine the immediately following bits in the bitstream, without actually removing the bits. To support this behavior, a '*' character can 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, we can check the value of next 32 bits in the bitstream as an unsigned integer without advancing the current position in the bitstream using the following representation:

```
aligned unsinged int (32)* next_code;
```

12.2.2 Variable Length Direct Representation Bit Fields

This case is covered by Rule E.1, by allowing the '*length'* field to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables. For example:

```
unsigned int(3) precision;
int(precision) DC;
```

12.2.3 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 one can extract the final desired value. This indirection can 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) is always 'bit'. The *output_type* entry is either a predefined type or a defined class (classes are defined in Subclause 12.3.1). 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 are specified as aggregates surrounded by curly braces, similar to C or C++ structures.

As an example, we have:

```
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 has to be identical to the type returned from the map. Example:

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, we can access a particular value of the map using the construct: `chroma_format.Ublocks`.

12.2.4 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 is 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 is 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.

This is illustrated in the following example (the class type 'val' is used, 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 asigned to the first element (`val.foo`), while the following 32 bits will 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 must match the type used in the class declaration associated with the map's return type.

12.3 Composite Data Types

12.3.1 Classes

Classes are the mechanism with which definition of composite types or objects is performed. Their definition is as follows.

Rule C.1: Classes
[aligned] [abstract] class *object_name* [extends *parent_class*] [: bit(*length*) [*id_name*]= *object_id* | *id_range*] {
[*element*; …] // zero or more elements
}

The different elements within the curly braces are definitions of elementary bitstream components as we saw in Subclause 12.2, or control flow that is discussed later on.

The optional 'extends *parent_class*' specifies that the class is "derived" from another class. Derivation means that all information present in the base class is also present in the derived class, and that all such information *precedes* in the bitstream any additional bitstream syntax declarations that are specified in the new class.

The *object_id* is optional, 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.

A derived class can appear at any point where its base class is specified in the bitstream. In order to be able to determine if the base class or one of its derived classes is present in the bitstream, the *object_id* (or range) is used. This identifier is given a particular value (or range of values) at the base class; all derived classes then have to specify their own unique values (or ranges of values) as well. If a class declaration does not provide an *object_id* then class derivation is still allowed, but any derived classes cannot substitute their base class in the bitstream. This mechanism expresses the concept of "polymorphism" in the context of bitstream syntax.

Examples:

```
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 bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, `foo.b` immediately precedes `foo.c` in the bitstream.

We can also encapsulate objects within other objects. In this case, the *element* mentioned at the beginning of this section is an object itself.

12.3.2 Abstract Classes

When the `abstract` keyword is used in the class declaration, it indicates that only derived classes of this class will be present in the bitstream. This implies that the derived classes can use the entire range of IDs available. The declaration of the abstract class requires a declaration of an ID, with the value 0. For 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
}
```

12.3.3 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. Because SDL follows a declarative approach, references to other objects 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 name and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a vlc, 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
}
```

12.3.4 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)'), *name* is the name of the array, and *length* is its length. For example, we can have:

```
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 .

12.3.5 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 can be given, but they must all agree on the type specification. This notation is also valid for multidimensional arrays. For example:

```
int(4) a[[3]][[5]];
```

indicates the element a(5, 3) of the array, 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 that ‘a[5]’ means that the array has five elements, whereas ‘a[[5]]’ implies that there are at least six.

12.3.6 Implicit Arrays

When a series of polymorphic classes is present in the bitstream, it can be represented as an array that has the type of the base class. Let us assume that a set of polymorphic classes is defined, derived from the base (abstract or not) class Foo:

```
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:

```
class Example {
   Foo f[]; // length implicitly obtained via ID resolution
}
```

To limit the minimum and maximum length of the array, a range specification can be inserted in the length:

```
class Example {
   Foo f[1 .. 255];  // at least 1, at most 255 elements
}
```

In this example, ‘f’ can have at least 1 and at most 255 elements.

12.4 Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C++ are allowed, including their precedence rules.

12.5 Non-Parsable Variables

In order to accommodate complex syntactic constructs in which context information cannot be directly obtained from the bitstream but is the 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 can 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
```

12.6 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 {
 …
}]

The following example illustrates the procedure.

```
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'. Another example is:

```
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 should 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).

12.7 Bult-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.

12.8 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.

13. Object Content Information

13.1 Introduction

Each media object that is associated with elementary stream data (i.e., described by an associated object descriptor), may have a separate OCI stream attached to it or may directly include a set of OCI descriptors as part of an `ObjectDescriptor` or `ES_Descriptor` as defined in Subclause 8.3.1.

The object descriptor for such media objects shall contain at most one ES_Descriptor referencing an OCI stream, identified by the value 0x0A for the `streamType` field of this ES_Descriptor.

Note: Since the scene description is itself conveyed in an elementary stream described by an object descriptor, it is possible to associate OCI to the scene as well.

OCI data is partitioned in access units as any media stream. Each OCI access unit corresponds to one OCI_Event, as described in the next subclause, and has an associated decoding time stamp (DTS) that identifies the point in time at which the OCI access unit becomes valid.

A pre-defined SL header configuration may be established for specific profiles in order to fit the OCI requirements in terms of synchronization.

13.2 Object Content Information (OCI) Syntax and Semantics

13.2.1 OCI Decoder Configuration

The OCI stream decoder needs to be configured to ensure proper decoding of the subsequent OCI data. The configuration data is specified in this subclause.

In the context of this Final Committee Draft of International Standard this configuration data shall be conveyed in the `ES_Descriptor` declaring the OCI stream within the `decoderSpecificInfo` class as specified in Subclause 8.3.5.

13.2.1.1 Syntax

```
class OCIStreamConfiguration {
   const bit(8) versionLabel = 0x01;
}
```

13.2.1.2 Semantics

`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.

13.2.2 OCI_Events

The Object Content Information stream is based on the concept of *events*. Each OCI_Event is conveyed as an OCI access unit that has an associated Decoding Time Stamp identifying the point in time at which this OCI_Event becomes effective.

13.2.2.1 Syntax

```
aligned(8) class OCI_Event : bit(16) event_id {
   unsigned int(16) length;
   bit(32) starting_time;
   bit(32) duration;
   OCI_Descriptor OCI_Descr[1 .. 255];
}
```

13.2.2.2 Semantics

`event_id` – contains the identification number of the described event.

`length` – gives the total length in bytes of the descriptors that follow.

`starting_time` – indicates the starting time of the event relative to the starting time 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.

Kommentar: Timing in OCI is disconnected from the rest of the MPEG-4 timing info. It would be nice if all timing information, except OCR/DTS, followed the same structure (e.g., SFTime).

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

`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.

Example: 02: 36:45:89 is coded as "0x023645" concatenated with "0b0101.1001".

`OCI_Descr[]` – is an array of one up to 255 `OCI_Descriptor` classes as specified in Subclause 13.2.3.2.

13.2.3 Descriptors

13.2.3.1 Overview

This subclause defines the descriptors that constitute the Object Content Information. A set of the descriptors may either be included in an `OCI_Event` or be part of an `ObjectDescriptor` or `ES_Descriptor` as defined in Subclause 8.3.1.

13.2.3.2 OCI_Descriptor Class

13.2.3.2.1 Syntax

```
abstract aligned(8) class OCI_Descriptor : bit(8) tag=0 {
}
```

13.2.3.2.2 Semantics

This class is a template class that is extended by the classes specified in the subsequent subclauses.

Note: *The consequence of this definition is that the OCI_Event class may contain a list of the subsequent descriptors in arbitrary order. The specific descriptor is identified by the particular tag value.*

13.2.3.3 Content classification descriptor

13.2.3.3.1 Syntax

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

13.2.3.3.2 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.

`length` – length of the remainder of this descriptor in bytes.

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.

13.2.3.4 Key Word Descriptor

13.2.3.4.1 Syntax

```
aligned(8) class KeyWordDescriptor extends OCI_Descriptor
            : bit(8) tag=KeyWordDescrTag {
   int i;
   unsigned int(8) length;
   bit(24) languageCode;
   unsigned int(8) keyWordCount;
   for (i=0; i<keyWordCount; i++) {
      unsigned int(8) keyWordLength;
      if (languageCode == latin) then {
         bit(8) keyWord[keyWordLength];
      } else {
         bit(16) keyWord[keyWordLength/2];
      }
   }
}
```

Kommentar: This must be changed to the actual code for 'latin'.

13.2.3.4.2 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.

length – length of the remainder of this descriptor in bytes.

languageCode – contains the ISO 639 [3] three character language code of the language of the following text fields.

Kommentar: This can be confusing. Since ISO 639 is cited, reference to Unicode here is superfluous (639 uses, I believe, ASCII characters – 7 bit – for the language code). I will remove it in the next pass unless somebody objects.

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

keyWordLength – specifies the length in bytes of each key word.

keyWord[] – a string that specifies the key word. Text information is coded using the Unicode character sets and methods [5].

13.2.3.5 Rating Descriptor

13.2.3.5.1 Syntax

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

13.2.3.5.2 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.

`length` – length of the remainder of this descriptor in bytes.

`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.

13.2.3.6 Language Descriptor

13.2.3.6.1 Syntax

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

13.2.3.6.2 Semantics

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

`length` – length of the remainder of this descriptor in bytes.

`languageCode` – contains the ISO 639 [3] three character language code of the corresponding audio/speech or text object that is being described.

13.2.3.7 Short Textual Descriptor

13.2.3.7.1 Syntax

```
aligned(8) class ShortTextualDescriptor extends OCI_Descriptor
            : bit(8) tag=ShortTextualDescrTag {
   unsigned int(8) length;
   bit(24) languageCode;
   unsigned int(8) nameLength;
   if (languageCode == latin) then {
      bit(8) eventName[nameLength];
      unsigned int(8) textLength;
      bit(8) eventText[textLength];
   } else {
      bit(16) eventName[nameLength/2];
      unsigned int(8) textLength;
      bit(16) eventText[textLength/2];
   }
}
```

Kommentar: The correct code should be included here. Is 'latin' the only case of 1-byte characters? I think Greek, for example, has single character encoding too (but I may be wrong).

13.2.3.7.2 Semantics

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

`length` – length of the remainder of this descriptor in bytes.

`languageCode` – contains the ISO 639 [3] three character language code of the language of the following text fields.

`nameLength` – specifies the length in bytes of the event name. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode [5].

`eventName[]`– a string that specifies the event name. Text information is coded using the Unicode character sets and methods [3].

`textLength` – specifies the length in byte of the following text describing the event. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode [5].

`eventText[]` – a string that specifies the text description for the event. Text information is coded using the Unicode character sets and methods [3].

13.2.3.8 Expanded Textual Descriptor

13.2.3.8.1 Syntax

```
aligned(8) class ExpandedTextualDescriptor extends OCI_Descriptor
            : bit(8) tag=ExpandedTextualDescrTag {
   int i;
   unsigned int(16) length;
   bit(24) languageCode;
   unsigned int(8) itemCount;
   for (i=0; i<itemCount; i++){
      unsigned int(8) itemDescriptionLength;
      if (languageCode == latin) then {
         bit(8) itemDescription[itemDescriptionLength];
      } else {
         bit(16) itemDescription[itemDescriptionLength/2];
      }
      unsigned int(8) itemLength;
      if (languageCode == latin) then {
         bit(8) itemText[itemLength];
      } else {
         bit(16) itemText[itemLength/2];
      }
   }
   unsigned int(8) textLength;
   int nonItemTextLength=0;
   while( textLength == 255 ) {
      nonItemTextLength += textLength;
      bit(8) textLength;
   }
   nonItemTextLength += textLength;
   if (languageCode == latin) then {
      bit(8) nonItemText[nonItemTextLength];
   } else {
      bit(16) nonItemText[nonItemTextLength/2];
   }
}
```

Kommentar: The correct code should be included here.

Kommentar: The correct code should be included here.

Kommentar: The correct code should be included here.

13.2.3.8.2 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.

`length` – length of the remainder of this descriptor in bytes.

`languageCode` - contains the ISO 639 [3] three character language code of the language of the following text fields.

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

`itemDescriptionLength` – specifies the length in byte of the item description. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode.

`itemDescription[]` – a string that specifies the item description. Text information is coded using the Unicode character sets and methods described in [5].

`itemLength` – specifies the length in byte of the item text. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode.

`itemText[]` – a string that specifies the item text. Text information is coded using the Unicode character sets and methods described in [3].

`textLength` – specifies the length in byte 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. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode.

`nonItemText[]` – a string that specifies the non itemised expanded text. Text information is coded using the Unicode character sets and methods described in [5].

13.2.3.9 Content Creator Name Descriptor

13.2.3.9.1 Syntax

```
aligned(8) class ContentCreatorNameDescriptor extends OCI_Descriptor
            : bit(8) tag= ContentCreatorNameDescrTag {
   int i;
   unsigned int(8) length;
   unsigned int(8) contentCreatorCount;
   for (i=0; i<contentCreatorCount; i++){
      bit(24) languageCode[[i]];
      unsigned int(8) contentCreatorLength[[i]];
      if (languageCode == latin) then {
         bit(8) contentCreatorName[[i]][contentCreatorLength];
      } else {
         bit(16) contentCreatorName[[i]][contentCreatorLength/2];
      }
   }
}
```

Kommentar: The correct code should be included here.

13.2.3.9.2 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.

`length` – length of the remainder of this descriptor in bytes.

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

`languageCode` – contains the ISO 639 [3] 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.

`contentCreatorLength[[i]]` – specifies the length in bytes of each content creator name. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode.

`contentCreatorName[[i]][]` – a string that specifies the content creator name. Text information is coded using the Unicode character sets and methods [3].

13.2.3.10 Content Creation Date Descriptor

13.2.3.10.1 Syntax

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

13.2.3.10.2 Semantics

This descriptor identifies the date of the content creation.

`length` – length of the remainder of this descriptor in bytes.

`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 Subclause 13.3). 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 start time is undefined all bits of the field are set to 1.

13.2.3.11 OCI Creator Name Descriptor

13.2.3.11.1 Syntax

```
aligned(8) class OCICreatorNameDescriptor extends OCI_Descriptor
            : bit(8) tag=OCICreatorNameDescrTag {
   int i;
   unsigned int(8) length;
   unsigned int(8) OCICreatorCount;
   for (i=0; i<OCICreatorCount; i++) {
      bit(24) languageCode[[i]];
      unsigned int(8) OCICreatorLength[[i]];
      if (languageCode == latin) then {
         bit(8) OCICreatorName[[i]][OCICreatorLength];
       } else {
         bit(16) OCICreatorName[[i]][OCICreatorLength/2];
      }
   }
}
```

Kommentar: The correct code should be included here.

13.2.3.11.2 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.

`length` – length of the remainder of this descriptor in bytes.

`OCICreatorCount` – indicates the number of OCI creators.

`languageCode[[i]]` – contains the ISO 639 [3] 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.

`OCICreatorLength[[i]]` – specifies the length in byte of each OCI creator name. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode.

`OCICreatorName[[i]]` – a string that specifies the OCI creator name. Text information is coded using the Unicode character sets and methods [5].

13.2.3.12 OCI Creation Date Descriptor

13.2.3.12.1 Syntax

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

13.2.3.12.2 Semantics

This descriptor identifies the creation date of the OCI description.

`length` – length of the remainder of this descriptor in bytes.

`OCICreationDate` - This 40-bit field contains the OCI creation date for the OCI data corresponding to the event in question, in Universal Time, Co-ordinated (UTC) and Modified Julian Date (MJD) (see Subclause 13.3). 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 start time is undefined all bits of the field are set to 1.

13.3 Conversion Between Time and Date Conventions (Informative)

The types of conversions that may be required are summarized in the diagram below.

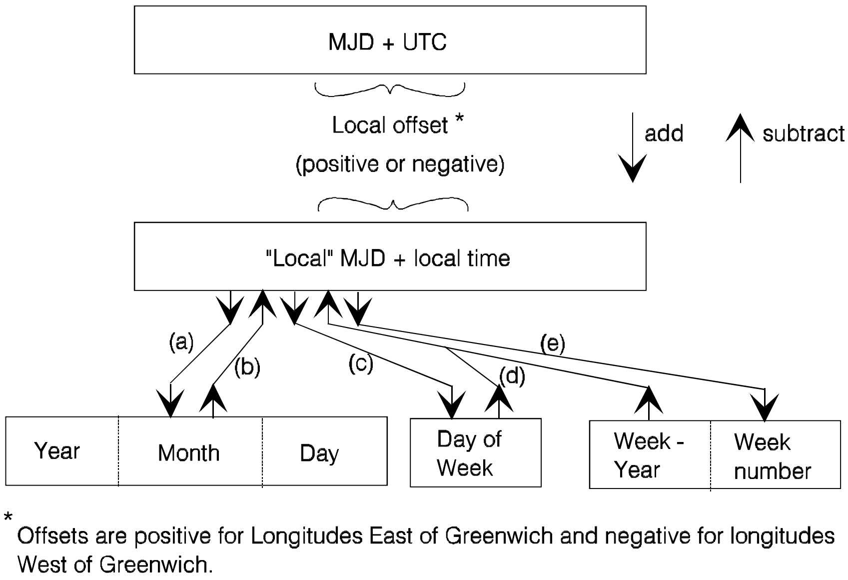


Figure 13-1: Conversion routes between Modified Julian Date (MJD) and Coordinated Universal Time (UTC)

The conversion between MJD + UTC and the "local" MJD + local time is simply a matter of adding or subtracting the local offset. This process may, of course, involve a "carry" or "borrow" from the UTC affecting the MJD. The other five conversion routes shown on the diagram are detailed in the formulas below.

Symbols used:

MJD:	Modified Julian Day
UTC:	Co-ordinated Universal Time
Y:	Year from 1900 (e.g. for 2003, Y = 103)
M:	Month from January (= 1) to December (= 12)
D:	Day of month from 1 to 31
WY:	"Week number" Year from 1900
MN:	Week number according to ISO 2015
WD:	Day of week from Monday (= 1) to Sunday (= 7)
K, L ,M' , W, Y':	Intermediate variables
×:	Multiplication
int:	Integer part, ignoring remainder
mod 7:	Remainder (0-6) after dividing integer by 7

a) To find Y, M, D from MJD

$Y' = \text{int} [(MJD - 15\,078{,}2) / 365{,}25]$

$M' = \text{int} \{ [MJD - 14\,956{,}1 - \text{int} (Y' \times 365{,}25)] / 30{,}6001 \}$

$D = MJD - 14\,956 - int(Y' \times 365{,}25) - int(M' \times 30{,}6001)$
If $M' = 14$ or $M' = 15$, then $K = 1$; else $K = 0$
$Y = Y' + K$
$M = M' - 1 - K \times 12$

b) To find MJD from Y, M, D

If $M = 1$ or $M = 2$, then $L = 1$; else $L = 0$
$MJD = 14\,956 + D + int[(Y - L) \times 365{,}25] + int[(M + 1 + L \times 12) \times 30{,}6001]$

c) To find WD from WJD

$WD = [(MJD + 2) \bmod 7] + 1$

d) To find MJD from WY, WN, WD

$MJD = 15\,012 + WD + 7 \times \{WN + int[(WY \times 1\,461 / 28) + 0{,}41]\}$

e) To find WY, WN from MJD

$W = int[(MJD / 7) - 2\,144{,}64]$
$WY = int[(W \times 28 / 1\,461) - 0{,}0079]$
$WN = W - int[(WY \times 1\,461 / 28) + 0{,}41]$

Example:	MJD	=	45 218	W	=	4 315
	Y	=	(19)82	WY	=	(19)82
	M	=	9 (September)	WN	=	36
	D	=	6	WD	=	1 (Monday)

Note: These formulas are applicable between the inclusive dates 1 900 March 1 to 2 100 February 28.

14. Profiles

This Subclause deinfes profiles of usage of Scene Description and Graphics primitives. These profiles are subsets of this specification to which system manufacturers and content creators can claim compliance in order to ensure interoperability.

The scene description profiles specify the scene description nodes and ROUTEs which are allowed to be present in the scene description. These profiles do not prescribe what media nodes are allowed in the scene descirption. This information is inferred from the Audio, Visual, and Graphics combination profiles.

The graphics combination profiles specify the graphics media nodes that are allowed to be present in a scene description.

Profile definitions alone are not sufficient to provide sufficient characterization of a terminal's capabilities. For this reason, within each profile a set of levels are defined. These levels constrain the values of various parameters so that an upper bound is identified, for use both by terminal manufacturers as well as content creators.

14.1 Parameters Used for Level Definitions

The following tables list restrictions for specific nodes and general restrictions that are used to limit a conformant scene's complexity. Where no explicit restriction is stated only general restrictions will apply to this node.

Kommentar: Shall we put this in the 2D profile section, and deal only with the scene description nodes since the capabilities of the media nodes are supposed to be dealt in the Media sub-groups ?

Table 14-1: Restrictions regarding Complexity for Grouping and Visual BIFS Nodes of 2D Profile

Node	Restriction	Remarks
Grouping Nodes		
Layout	restricted by #nodes and #streams and depth of scene graph	Unclear definition in spec.: • dependency of scrollRate field to bitrate for StreamingText unclear (should be the same!) • only used for formatting of Text and StreamingText? (proposed solution: yes!) • multiple children nodes allowed? (proposed solution: no! - if yes: how are they handled?)
Transform2D	#transformations to be distinguished by • scaling • rotation • translation	
Group		no inherent complexity, only limited by general limitations (e.g. max. #nodes and scene depth)
Inline2D	Either all inlined scene graphs jointly match a conformance point OR each stream individually matches a conformance point	
Shape2D Appearance		'container' nodes, no inherent complexity
Appearance Nodes		
VideoObject	#nodes according to the selected Video Combination Profile	
MovieTexture ImageTexture	#pixels to be transformed for all MovieTexture and ImageTexture nodes	It is assumed that the VideoObject2D and Image2D nodes cannot be transformed and mapped onto a specific geometry
MovieTexture ImageTexture VideoObject2D Image2D Background2D	#pixels to be held in memory for all these nodes	This may be a superset of a Visual Combination Profile since it includes moving Visual Objects and still images.
TextureTransform	limited by #pixels to be transformed (see MovieTexture and ImageTexture nodes)	
Material2D	transparency allowed/not allowed	The transparency field may cause additional complexity! → restriction tbd
ShadowProperties	allowed/not allowed	further/better restriction of this node depends on clarifications in spec.: • to which (geometry) nodes may this node be applied? • How shall this node be rendered? (The shadow's color will depend on the material where the shadow falls on!)
LineProperties	allowed/not allowed	further/better restriction of this node depends on clarifications in spec.: • to which (geometry) nodes may this node be applied?

Node	Restriction	Remarks
Geometry Nodes		
Circle	#nodes	
Rectangle	#nodes	
Text	#characters for all Text (including buffer for StreamingText???) nodes	The 'maxExtent' may lead to a rather high rendering complexity of this node. (→ adequate restriction tbd)
StreamingText		A box size needs to be defined in which the text (or parts of it) is to be displayed. It is assumed that this box is defined by a parent Layout node. (→ corresponding clarification in spec. is required!)
FontStyle	#of supported font styles.	The 3 generic style families SERIF, SANS, and TYPEWRITER will be enough in many cases
IndexedFaceSet	total #faces for all IndexedFaceSet nodes	Texture mapping on IndexedFaceSet nodes potentially causes additional complexity (besides the computational requirements for transformation processing) due to a high amount of transformation matrices (one per face! → memory requirements!)
TextureCoordinate	#nodes	may lead to an additional complexity of the IndexedFaceSet node (see remarks of that node)
IndexedLineSet	total #lines for all IndexedLineSet nodes	
PointSet2D		restricted only by limitations of Color and Coordinate2D nodes
Curve2D	total #points for all Curve2D nodes	
Misc. Nodes		
Color	total #color fields (1 color field = 3 values: [R,G,B]) for all color nodes	#SFColor values (as used by the ShadowProperties and LineProperties nodes) might have to be included in this restriction
Coordinate2D	#point fields (1 point field = 2 values: [x,y]) for all Coordinate2D nodes	
Composite2DTexture	#nodes	The implications of this node are not well understood. Nesting of these nodes should not be allowed. Its size must be added to the overall #pixels restriction.
AnimationStream	#fields to be changed per second	This increases system control overhead

Table 14-2: General Complexity Restrictions for 2D Profile

General	#nodes	to limit the necessary memory size for all nodes
	#streams	limited also by max. #object descriptors
	#object descriptors	
	sum of all bounding boxes' surfaces < F_{max} F_{max} tbd	to limit the maximum number of (texture) pixels to be rendered

	display size (CIF, QCIF, etc.)	e.g. to limit the max. #pixels to be transformed
	color space	to limit the maximum number of allowed colors
	depth of scene graph	limits the number of cascaded transforms

14.2 Scene Description Profiles

The following scene description profiles are defined:
1. Simple
2. 2D
3. VRML
4. Audio
5. Complete

14.2.1 Simple Profile

The simple profile profiles for the minimal set of tools necessary to place one or more media objects in a scene. It is intended for applications such as the emulation of traditional broadcast TV.

The following table summarizes the tools included in the profile.

Simple Profile			
Nodes	**ROUTEs**	**BIFS Animation**	**BIFS Updates**
Layer2D, Transform2D	No	No	Yes

In addition, the following restrictions apply:

1. No rotation or scaling is allowed in the transform nodes (if present, their values will be ignored).
2. Only pixel metrics are to be used (i.e., the ‘isPixel’ parameter should be set to 1).
3. No childrenLayer children nodes are allowed in the Layer2D node.

14.2.2 2D Profile

The 2D profiles provides 2D scene description functionalities, and is intended for applications supporting features such as 2D transformations and alpha blending.

The following table summarizes the tools included in the profile.

2D Profile			
Nodes	**ROUTEs**	**BIFS Animation**	**BIFS Updates**
Shared and 2D nodes, and the Layer2D node	Yes	Yes	Yes

14.2.3 VRML Profile

The VRML profiles provides scene description elements that are common to this Final Committee Draft of International Standard and ISO/IEC 14772-1 (VRML). It is indended for applications where reuse of existing systems or content that is conformant to one of these specifications is important.

The following table summarizes the tools included in the profile.

VRML Profile			
Nodes	**ROUTEs**	**BIFS Animation**	**BIFS Updates**
Common VRML/MPEG-4 scene description nodes	Yes	No	Yes

14.2.4 Audio Profile

The Audio profile provides only the audio-related scene description nodes. It is intended for applications that use only audio features, such as radio broadcast.

The following table summarizes the tools included in the profile.

Audio Profile			
Nodes	**ROUTEs**	**BIFS Animation**	**BIFS Updates**
Audio scene description nodes	Yes	No	Yes

14.2.5 Complete Profile

The Complete Scene Description Profile provides the complete set of scene description capabilities specified in Subclause 7.2 the Final Committee Draft. It is intended for applications that will use the whole spectrum of scene description features made available by this specification like virtual world with streaming media.

The following table summarizes the tools included in the profile.

Complete Profile			
Nodes	**ROUTEs**	**BIFS Animation**	**BIFS Updates**
All	Yes	Yes	Yes

14.3 Graphics Combination Profiles

Graphics combination profiles define the set of graphics primitives (graphics media nodes) which are allowed to be present in a scene description. Two such profiles are defined:

1. 2D
2. Complete

14.3.1 2D Profile

The 2D graphics combination profiles involves all 2D graphics media nodes, and is intended for use in conjuction with the 2D or Complete scene description profiles.

2D Profile
Nodes
Shared and 2D graphics media nodes

14.3.2 Complete Profile

The complete profile includes all graphics media nodes, and is intended for use in conjuction with the VRML or Complete scene description profiles.

Complete Profile
Nodes
All

15. Elementary Streams for Upstream Control Information

Media objects may require upstream control information to allow for interactivity.

The content creator needs to advertise the availability of an upstream control stream by means of an `ES_Descriptor` for this stream with the `upstream` flag set to one. This `ES_Descriptor` shall be part of the same object descriptor that declares the downstream elementary stream for this media object. See Subclause 8.3 for the specification of the object descriptor syntax.

An elementary stream flowing from receiver to transmitter is treated the same way as any downstream elementary stream. The ES data will be conveyed through the Elementary Stream Interface to the sync layer where the access unit data is packaged in SL packets. The parameters for the sync layer shall be selected as requested in the ES_Descriptor that has advertised the availability of the upstream control stream.

The SL-packetized stream (SPS) with the upstream control data is subsequently passed through the Stream Multiplex Interface to a delivery mechanism similar to the downstream SPS's. The interface to this delivery mechanism may be embodied by the DMIF Application Interface as specified in Part 6 of this Final Committee Draft of International Standard.

Note: The content of upstream control streams is specified in the same part of this specification that defines the content of the downstream data for this media object. For example, control streams for video compression algorithms are defined in 14496-2.

Annex A: Bibliography

[1] A. Eleftheriadis, "Flavor: A Language for Media Representation," *Proceedings, ACM Multimedia '97 Conference*, Seattle, Washington, November 1997, pp. 1–9.

[2] C. Herpel, "Elementary Stream Management in MPEG-4," *IEEE Trans. on Circuits and Systems for Video Technology*, 1998 (to appear).

[2] Flavor Web Site, http://www.ee.columbia.edu/flavor.

[3] R. Koenen, F. Pereira, and L. Chiariglione, "MPEG-4: Context and Objectives," *Signal Processing: Image Communication*, Special Issue on MPEG-4, Vol. 9, Nr. 4, May 1997.

[4] F. Pereira, and R. Koenen, "Very Low Bitrate Audio-Visual Applications," *Signal Processing: Image Communication*, Vol. 9, Nr. 1, November 1996, pp. 55-77.

[5] A. Puri and A. Eleftheriadis, "MPEG-4: An Object-Based Multimedia Coding Standard Supporting Mobile Application," *ACM Mobile Networks and Applications Journal*, 1998 (to appear).

Annex B: Time Base Reconstruction (Informative)

B.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 sender's clock at the receiver, when required (e.g., for broadcast applications). A normative method for this reconstruction is not specified. The following describes the process only at a conceptual level.

B.1.1 Adjusting the Receiver's OTB

Each elementary stream may be generated by an encoder with a different object time base (OTB). For each stream that conveys OCR information, it is possible for the receiver to adjust a local OTB to the encoders' OTB. This is done by well-known PLL techniques. The notion of time for each object can therefore be recovered at the receiver side.

B.1.2 Mapping Time Stamps to the STB

Kommentar: I think the text here is incorrect. I left the text as is for now, but we have an AHG set up (co-chaired by Carsten and me) that will review it between May 15 and the Dublin meeting.

The OTBs of all objects may run at a different speed than the STB. 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} in terms of STB of a composition unit can be calculated from the composition time stamp value t_{OPT} in terms of the relevant OTB being transmitted 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 media object's OTB, conveyed by an OCR
$t_{STB-START}$	value of receiving terminal's STB when the first OCR time stamp of the media object is encountered
$t_{OTB-START}$	value of the first OCR time stamp of the media object

$$\Delta t_{OTB} = t_{OTB} - t_{OTB-START}$$

$$\Delta t_{STB} = t_{STB} - t_{STB-START}$$

The quotient $\Delta t_{STB} / \Delta t_{OTB}$ is the scaling factor between the two time bases. In cases where the clock speed and resolution of the media encoder 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 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 also have to be mapped with the same mechanism.

With this mechanism it is possible to synchronize to several OTBs so that correct decoding and composition of composition units from several media objects is possible.

B.1.3 Adjusting the STB to an OTB

When all media objects in a session use the same OTB, it is possible to lock the STB to this OTB by well-known PLL techniques. In this case the mapping described in the previous subclause is not necessary.

B.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 decoder but not in applications that require flow-control. For example, file-based playback does not require time base reconstruction and hence time stamps alone are sufficient for synchronization.

In the absence of time stamps, the decoder 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 of the terminal's behavior.

In the case that a universal clock is available that can be shared between peer terminals, it may be used as a common time base and it thus enable the use the Systems Decoder Model without explicit OCR transmission. The mechanisms for doing so are application-dependent and are not defined in this specification.

B.2 Temporal aliasing and audio resampling

A terminal compliant with this Final Committee Draft of International Standard is not required to synchronize decoding of AUs and composition of CUs. In other words, the STB does not have to be identical to the OTB of any media object. The number of decoded and actually presented (displayed/played back) units per second may therefore differ. Hence, temporal aliasing may occur, resulting from composition units being presented multiple times or being skipped.

If audio signals are encoded on a system with an OTB different from the STB of the decoder, even nominally identical sampling rates of the audio samples will not match, 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.

B.3 Reconstruction of a Synchronised Audiovisual Scene: A Walkthrough

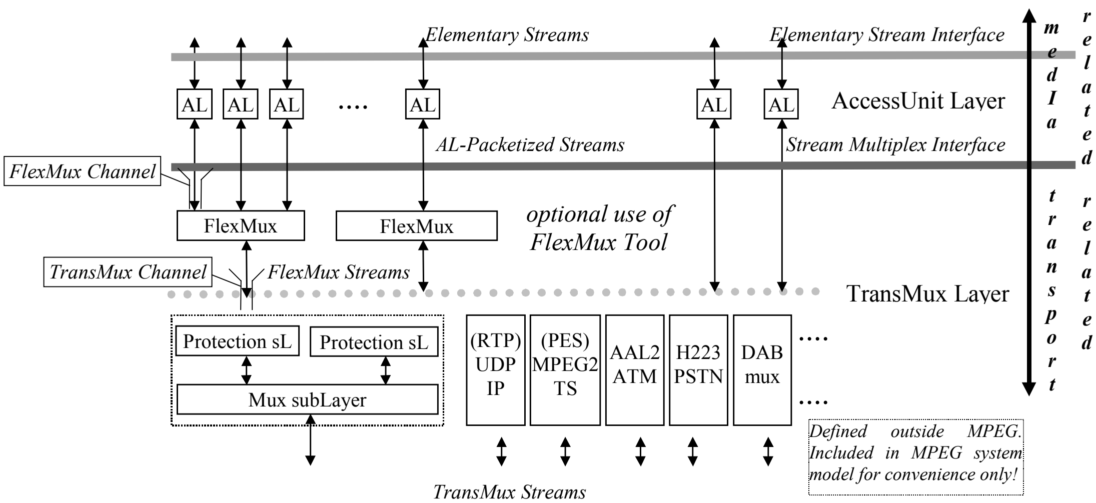
The different steps to reconstruct a synchronized scene are as follows:

1. The time base for each object is recovered either from the OCR conveyed with the SL-packetized elementary stream of this object or from another object present in the scene graph.
2. Object time stamps are mapped to the STB 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 media object decoder at its implicit or explicit DTS and the resulting one or more composition units are placed in the composition memory,
5. The compositor may access each CU between its CTS and the CTS of the subsequent CU.

Annex C: Embedding of SL-Packetized Streams in TransMux Instances (Informative)

The specification of this Final Committee Draft of International Standard terminates at the Stream Multiplex Interface, which is an interface at which packets of elementary stream data are conveyed to a receiving terminal. This assumes that a multitude of transport protocol stacks exists that are able to transfer the packetized elementary stream data end-to-end.

The generic term TransMux Layer is used to abstract all potential protocol stacks that supply a transport multiplex for content compliant with this Final Committee Draft of International Standard, as shown in the following figure. The FlexMux tool specified in Subclause 11.2 may optionally be used as the first element of any of these protocol stacks.



This informative annex presents a number of examples on how content complying with this Final Committee Draft of International Standard can be embedded in specific instances of such a TransMux.

Note: *The examples below have been implemented and verified to some degree by individual parties. They are provided here as starting point for possible standardisation by interested parties in the appropriate context and are pending complete verification.*

C.1 ISO/IEC 14496 content embedded in ISO/IEC 13818-1 Transport Stream

C.1.1 Introduction

This informative annex describes an encapsulation method for a stream complying to this Final Committee Draft of International Standard by an ISO/IEC 13818-1 Transport Stream [1]. This informative annex describes a way to transmit the InitialObjectDescriptor, ObjectDescriptorStream, SceneDescriptionStream and audiovisual elementary stream data specified in Parts 2 and 3 of this Final Committee Draft of International Standard in an 13818-1 Transport Stream. This informative annex also describes a method on how to specify the 14496 indication in the 13818-1 Program Specific Information.

C.1.2 ISO/IEC 14496 Stream Indication in Program Map Table

In a 13818-1 Transport Stream, the elementary streams are transmitted in TS packets. The Packet ID (PID) identifies to which stream the payload of a TS packet belongs. Some specific PID values are defined in Table 2-3 of ISO/IEC 13818-1. The value '0' of the PID is used for the Program Association Table (PAT) specified in Subclause 2.4.4.3 of ISO/IEC 13818-1. The PAT specifies one or more PIDs that convey Program Map Tables (PMT) as specified in Subclause 2.4.4.8 of ISO/IEC 13818-1 for one or more programs. A PMT consists of one or more TS_program_map_sections as specified in Table 2-28 of ISO/IEC 13818-1. The PMT contains for each elementary stream that is part of one program information such as the stream_type and Elementary_PID. The stream_type specified in Table 2-29 of ISO/IEC 13818-1 identifies the data type of the stream. The Elementary_PID identifies the PID of the TS packets that convey the data for this elementary stream.

ISO/IEC FCD 14496 content is accessed through an InitialObjectDescriptor (InitialOD) specified in Subclause 8.3.2. For content access, the InitialOD must first be acquired at the receiving terminal as described in Subclause 8.4.3. The InitialOD contains one or more ES_Descriptors with `streamType` of SceneDescriptionStream and `streamType` of ObjectDescriptorStream. These descriptors allow to locate the respective streams using an additional stream map table.

When an ISO/IEC 13818-1 program contains ISO/IEC 14496 elementary streams, the TS_program_map_section shall have the description of "MPEG-4" stream_type in the second loop. In this loop, the Elementary_PID field specifies the PID of the TS packets that convey the InitialOD. Thus, the pointer to the ISO/IEC FCD 14496 content is indicated in an MPEG-2 program.

C.1.3 Object Descriptor Encapsulation

Each ISO/IEC 14496 elementary stream has an associated ES_Descriptor as part of an ObjectDescriptor specified in Subclause 8.3.3. The ES_Descriptor contains information of the associated elementary stream such as the decoder configuration, SL packet header configuration, IP information etc.

The InitialObjectDescriptor is treated differently from the subsequent ObjectDescriptors, since it has the content access information and it has to be retrieved as the first ObjectDescriptor at the receiving terminal.

C.1.3.1 InitialObjectDescriptorSection

The InitialOD shall be retransmitted periodically in broadcast applications to enable random access, and it must be received at the receiving terminal without transmission errors. For these purposes, the transmission of the InitialOD in InitialObjectDescriptorSections (InitialOD-Section) is defined here.

The InitialOD-Section is based on the private section specified in Subclause 2.4.4.10 of ISO/IEC 13818-1. In the InitialOD-Section, the section_syntax_indicator is always set to '1' in order to use the table_id_extension, version_number, current_next_indicator, section_number, last_section_number and CRC_32. The InitialOD-Section provides filtering and error detection information. The version_number field is used to discard a section at the receiver terminal when the terminal has already received a section labeled by the same version_number. The CRC_32 is used to detect if the section contains errors.

The InitialOD-Section contains an InitialOD and a StreamMapTable. The StreamMapTable is specified in Subclause C.1.3.3.

The InitialOD-Section is transmitted in TS packets in the same way as other PSI sections specified in ISO/IEC 13818-1. The PID of the TS packets that convey the InitialOD is identified by the PMT, using stream_type of "MPEG-4" as described in Subclause C.1.2.

C.1.3.1.1 Syntax

```
class InitialObjectDescriptorSection {
   bit(8)  table_id;
   bit(1)  section_syntax_indicator;
   bit(1)  private_indicator;
```

```
   const bit(2)  reserved=0b11;
   bit(12)  private_section_length;
   bit(16)  table_id_extension;
   const bit(2)  reserved=0b11;
   bit(5)  version_number;
   bit(1)  current_next_indicator;
   bit(8)  section_number;
   bit(8)  last_section_number;
   StreamMapTable  streamMapTbl;
   InitialObjectDescriptor  initialObjDescr;
   bit(32)  CRC_32;
}
```

C.1.3.1.2 Semantics

`table_id` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_syntax_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1 and it is always set to '1'.

`private_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`private_section_length` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1. Due to the bit length limitation of this field, the sum of the `streamMapTbl` and `initiObjDescr` fields in this section shall not exceed 4084 bytes.

`table_id_extension` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`version_number` – is incremented by 1 modulo 32 when the a `streamMapTbl` or an `initialObjDescr` is changed.

`current_next_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`last_section_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`streamMapTbl` – is a StreamMapTable as specified in Subclause C.1.3.3. The `streamMapTbl` shall contain a list of the ES_IDs of all elementary streams that are refered to by the subsequent `initialObjDescr` and associate each of them to the PID and FlexMux channel, if applicable, that conveys the elementary stream data.

`initialObjDescr` – is an InitialObjectDescriptor as specified in Subclause 8.3.2.

`CRC_32` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

C.1.3.2 ObjectDescriptorStreamSection

ObjectDescriptors are conveyed in an ObjectDescriptorStream as specified in Subclause 7.3.2. The sync layer, specified in Subclause 10.2, may divide the ObjectDescriptorStream access units into more than one SL packet. The `accessUnitStartFlag` in the SL packet header is used to identify if an access unit starts in this packet as defined in Subclause 8.2.4. Similarly, the `randomAccessPointFlag` is used to identify a random access point.

The ObjectDescriptorStream shall be retransmitted periodically to enable random access, and it must be received at the receiving terminal without transmission errors. For these purposes, the transmission of an ObjectDescriptorStream in ObjectDescriptorStreamSections (ODStream-Section) is defined here.

The ODStream-Section is similar to the InitialOD-Section. However, the presence of the StreamMapTable and the subsequent SL packet are optional. The StreamMapTable is specified in Subclause C.1.3.3. The SL packet contains a complete or a part of an access unit of the ObjectDescriptorStream.

The size of an ODStream-Section shall not exceed 4096 bytes. Up to 256 ODStream-Sections may share the same `version_number` and, hence, form one version. Each version shall allow random access to a complete set of ObjectDescriptors as they are required at the current time.

The ODStream-Section is transmitted in TS packets in the same way as other PSI sections specified in ISO/IEC 13818-1. The PID of the TS packets that convey the ObjectDescriptorStream is identified by the StreamMapTable in the InitialOD-Section.

C.1.3.2.1 Syntax

```
class ObjectDescriptorStreamSection {
   bit(8)  table_id;
   bit(1)  section_syntax_indicator;
   bit(1)  private_indicator;
   const bit(2)  reserved=0b11;
   bit(12)  private_section_length;
   bit(16)  table_id_extension;
   const bit(2)  reserved=0b11;
   bit(5)  version_number;
   bit(1)  current_next_indicator;
   bit(8)  section_number;
   bit(8)  last_section_number;
   const bit(6)  reserved=0b1111.11;
   bit(1)  streamMapTblFlag;
   bit(1) OD_PacketFlag;
   if (streamMapTblFlag)
      StreamMapTable  streamMapTbl[[section_number]];
   if (OD_PacketFlag)
      SL_Packet  objDescrStrmPacket[[section_number]];
   bit(32)  CRC_32;
}
```

C.1.3.2.2 Semantics

`table_id` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_syntax_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1 and it is always set to ‘1’.

`private_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`private_section_length` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1. Due to the bit length limitation of this field, the sum of the `streamMapTbl` and `objDescrStrm` fields in this section shall not exceed 4083 bytes.

`table_id_extension` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`version_number` – is incremented by 1 modulo 32 when the content of a `streamMapTbl` or an `objDescrStrmPacket` in one of the sections that constitute this version is changed.

`current_next_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`last_section_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`streamMapTblFlag` – if set to ‘1’ indicates the `streamMapTbl` field will follow.

OD_PacketFlag – if set to ‘1’ indicates that an `objDescrStrmPacket` will follow.

`streamMapTbl[]` – is a StreamMapTable as specified in Subclause C.1.3.3. At least one `streamMapTbl` shall exist in the set of ODStream-Sections that constitute one version of an object descriptor stream. The set of streamMapTables in one such version shall map the complete list of all ES_IDs that are refered to in this object descriptor stream to their respective PIDs and FlexMux channels, if applicable.

`objDescrStrmPacket[]` – is an SL packet that contains a complete or a part of an access unit of the ObjectDescriptorStream.

`CRC_32` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

C.1.3.3 StreamMapTable

The StreamMapTable associates elementary stream descriptors contained in object descriptors with the actual channels in which the streams are carried. The association is performed based on the ES_ID number contained in the elementary stream descriptor., and indicates the PID that contains the stream as well as the FlexMux channel number (when FlexMux is used).

The StreamMapTable is a list of the ES_ID, FlexMux channel number and PID within this transport stream for each elementary stream within the name scope of this object descriptor stream. The StreamMapTable is located in the InitialOD-Section specified in Subclause C.1.3.1 and the OD-Section specified in Subclause C.1.3.2.

C.1.3.3.1 Syntax

```
class StreamMapTable () {
   const bit(6)  reserved=0b1111.11;
   bit(10)  streamCount;
   for (i=0; i<streamCount; i++) {
      bit(16)  ES_ID;
      bit(1)  FlexMuxFlag;
      if (FlexMuxFlag) {
         bit(8)  FlexMuxChannel;
      }
      const bit(2) reserved=0b11;
      bit(13)  ES_PID;
   }
}
```

C.1.3.3.2 Semantics

`streamCount` – is the number of streams for which the stream association is conveyed in this table.

`ES_ID` – provides a unique label for each elementary stream as defined in Subclause 8.3.3.

`FlexMuxFlag` – if set to '1' indicates the presence of the `FlexMuxChannel` field. If this flag is set to '0', it indicates the FlexMux specified in Subclause 11.2 is not used.

`FlexMuxChannel` – is the FlexMux channel number for this stream.

`ES_PID` – is the PID in the Transport Stream that conveys the elementary stream.

C.1.4 Scene Description Stream Encapsulation

A SceneDescriptionStream as defined in this Final Committee Draft of International Standard may convey two types of information.. The first is BIFS-Commands, specified in Subclause 9.2.17.2, which conveys the necessary information to compose the audiovisual objects in a scene. The second one is BIFS-Anim, specified in Subclause 9.2.17.3, which is used to transmit updated parameters for the various fields of specific (user-specified) BIFS nodes. The type of SceneDescriptionStream is identified in the decoderSpecificInfo specified in Subclause 8.3.5 as part of the ES_Descriptor for this stream.

Only encapsulation of BIFS-Command streams is considered here, since the BIFS-Anim stream is just another elementary stream, that shall be encapsulated in the same way as other audiovisual elementary streams as described in Subclause C.1.5.

C.1.4.1 BIFSCommandSection

The BIFS-Command structure contains one or more update commands to either convey an entire scene or perform a partial update. The SceneReplaceCommand, in particular, specifies a complete scene.

The sync layer divides the BIFS-Commands into one or more access units. An access unit contains one or more BIFS-Commands that become valid at the same time in the receiver terminal. An access unit may be divided into more than one SL packets. The `accessUnitStartFlag` in the SL packet header is used to identify if an access unit starts at this packet.

If the `randomAccessPointFlag` in the SL packet header is set to '1', an access unit shall have the SceneReplaceCommand. Otherwise, the randomAccessPointFlag shall be set to '0'.

The SceneReplaceCommand shall be retransmitted periodically to enable random access, and it must be received at the receiving terminal without transmission errors. For these purposes, the transmission of SceneDescriptionStreams with BIFS-Command information in BIFSCommandSections is defined here.

The BIFSCommandSection is based on the private section specified in Subclause 2.4.4.10 of ISO/IEC 13818-1. In the BIFSCommandSection, the section_syntax_indicator is fixed to '1'. Therefore, optional fields such as the table_id_extension, version_number, current_next_indicator, section_number, last_section_number and CRC_32, are used. The BIFSCommandSection provides filtering and error detection information as well as the InitialOD-Section and ODStream-Section.

The size of a BIFSCommandSection shall not exceed 4096 bytes. Up to 256 BIFSCommandSections may share the same `version_number` and, hence, form one version. Each version shall allow random access to a complete set of BIFS-Command information as it is required at the current time.

The BIFSCommandSection is transmitted in TS packets in the same way as other PSI sections specified in ISO/IEC 13818-1. The PID of the TS packets that convey the BIFSUpdates is identified by the StreamMapTable in the InitialOD-Section or ODStream-Section.

C.1.4.1.1 Syntax

```
class BIFSCommandSection {
   bit(8)  table_id;
   bit(1)  section_syntax_indicator;
   bit(1)  private_indicator;
   const bit(2)  reserved=0b11;
   bit(12)  private_section_length;
   bit(16)  table_id_extension;
   const bit(2)  reserved=0b11;
   bit(5)  version_number;
   bit(1)  current_next_indicator;
   bit(8)  section_number;
   bit(8)  last_section_number;
   SL_Packet  BIFSCommandFrmPacket[[section_number]];
   bit(32)  CRC_32;
}
```

C.1.4.1.2 Semantics

`table_id` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_syntax_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1 and it is always set to '1'.

`private_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`private_section_length` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`table_id_extension` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`version_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`current_next_indicator` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

`section_number` – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

last_section_number – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

BIFSCommandFrm Packet[] – is an SL packet that contains a complete or a part of an access unit of the BIFS-Commands. The size of BIFSCommandFrmPacket[] shall not exceed 4084 bytes.

CRC_32 – is defined in Subclause 2.4.4.11 of ISO/IEC 13818-1.

C.1.5 Audiovisual Stream Encapsulation

Representations of audio and visual objects, defined in Parts 2 and 3 of this Final Committee Draft of International Standard, as well as the BIFS-Anim information, are conveyed in separate elementary streams. The sync layer divides each elementary stream into one or more access units. An access unit may be divided into more than one SL packets. Furthermore, SL packets of different elementary streams may be conveyed through the same PID by using FlexMux.

The alignment of SL packet and FlexMux packet to the TS packet payload is defined here, since neither packets's header syntax provides for a sync-word. The SL packet or FlexMux packet shall start at the first byte of the TS packet payload if the payload_unit_start_indicator specified in Subclause 2.4.3.2 of ISO/IEC 13818-1 is set to '1'. If the payload_unit_start_indicator is set to '0', no start of an SL packet or FlexMux packet is signaled in this TS packet.

When the FlexMux is not used, a TS packet shall not contain more than one SL packet. In this case, a TS packet shall contain a complete or a part of the SL packet. When the FlexMux is used, a TS packet may contain more than one FlexMux packet.

PIDs within the transport stream that convey audiovisual elementary streams are identified by the StreamMapTable in the ODStreamSection. The PID within the transport stream that conveys a BIFS-Anim stream is identified by the StreamMapTable in the InitialODSection or ODStreamSection.

C.1.6 Framing of SL Packets and FlexMux Packets into TS Packets

When an SL packet or one or more FlexMux packets do not fit the size of a TS packet, stuffing is required. Two methods for stuffing are specified here. The first one uses the adaptation field specified in Subclause 2.4.3.4 of ISO/IEC 13818-1. The second one uses the paddingFlag and paddingBits in the SL packet header.

C.1.6.1 Use of MPEG-2 TS Adaptation Field

The adaptation field specified in Subclause 2.4.3.4 of ISO/IEC 13818-1 is used when the size of the SL packet or FlexMux packet(s) does not fit the size of the TS packet payload. The size of the adaptation field is identified by the adaptation_field_length field. The adaptation_field_length is an 8-bit field and specifies the length of adaptation field immediately following the adaptation_field_length. The length of the adaptation field can be calculated as follows :

- in case of one or more FlexMux packets:

  adaptation_field length = 188 - 4 - MIN [FlexMux packet(s) length, 184]

- in case of an SL packet:

  adaptation_field length = 188 - 4 - MIN [SL packet length, 184]

C.1.6.2 Use of MPEG-4 PaddingFlag and PaddingBits

The paddingFlag and paddingBits are the fields that are used in the SL packet header to fit the FlexMux packets into TS packets. These fields allow to construct a FlexMux packet that contains padding bytes only. This type of FlexMux packet can be used to adjust the size of TS packet to 188 bytes. An example is shown in Figure C-1.

Note: *This method cannot be used when FlexMux is not used for this elementary stream or when the size of the FlexMux packet equals 183 bytes. In that case, the method described in Subclause C.1.6.1 can be applied.*

The packets are built as follows :

- TS_packet = TS_packet_header (4 bytes) + TS_packet_payload
- TS_packet_payload = FlexMuxPacket
- FlexMuxPacket = FlexMuxPacketHeader + FlexMuxPacketPayload
- FlexMuxPacketPayload= SL_Packet
- SL_Packet = SL_PacketHeader + SL_PacketPayload
- SL_PacketPayload = 1 SegmentOfAU
- AU = Σ (SegmentOfAU)
-

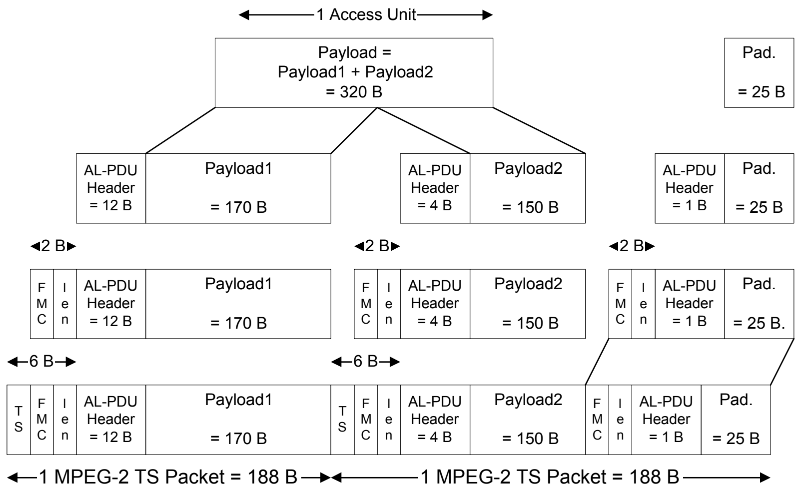


Figure C-1: An example of stuffing for the MPEG-2 TS packet

C.2 ISO/IEC 14496 Content Embedded in ISO/IEC 13818-6 DSM-CC Data Carousel

C.2.1 Introduction

The purpose of this informative annex is to describe a method based on the Data Carousel scenario of the ISO/IEC 13818-6 DSM-CC Download Protocol to provide FlexMux-like functionality for ISO/IEC 13818-1 Transport Streams. The resulting FlexMux is called the ISO/IEC 13818-1 FlexMux. The DSM-CC Data Carousel scenario of the Download Protocol is adjusted to support the transmission of SL packets as module blocks. Each module is considered as a FlexMux channel. Advantages of the method are:

- Compatibility with existing MPEG-2 Hardware. The new MPEG-4 services are transported in DSM-CC sections which do not interfere with the current delivery of MPEG-2 Transport of Video and Audio streams. Consequently, current receiver hardware will not be affected by the new design.
- Use of current decoder hardware filtering capabilities to route MPEG-4 elementary streams to separate object buffers.

- Easy management of FlexMux channels. New channels can be added or removed easily by simply updating and changing the version of the carousel directory message.
- Support of several channels within one PID. As a result, maintaining the Stream Map Table becomes much easier as it does not depend on hard coded PID values.

C.2.2 DSM-CC Data Carousel

The Data Carousel scenario of the DSM-CC Download Protocol is specified in the ISO/IEC 13818-6 International Standard. It is built on the DSM-CC download protocol where data and control messages are periodically re-transmitted following a pre-defined periodicity.

The download server sends periodic download control messages which allow a client application to discover the data modules being transmitted and determine which, if any, of these modules must be acquired.

The client typically retrieves a subset of the modules described in the control messages. The data modules are transmitted in blocks by means of download data messages. Acquisition of a module does not necessarily have to start at the first block in the module as download data messages feature a block numbering field that allows any client application to assemble the blocks in order.

When the Data Carousel is encapsulated in a TS, special encapsulation rules apply to facilitate acquisition of the modules. The DSM-CC Broadcast Data Carousel framework allows many data modules to be transported within a single PID. Therefore small and large data modules can be bundled together within the same PID.

C.2.3 ISO/IEC 13818-1 FlexMux Overview

The DSM-CC Data Carousel scenario of the Download protocol supports the transmission of unbounded modules or modules of unspecified size. The module size of such modules is set to 0**.** Each unbounded module can be viewed as an ISO/IEC 14496-1 FlexMux channel. Since modules are transported in chunks by means of DownloadDataBlock messages, a 13818-1 FlexMux SL packet is defined to correspond to the payload of a DownloadDataBlock message. The FlexMux channel number is the identifier of the module. The FlexMux channels are identified by DSM-CC sections for which table_id is equal to 0x3B (these sections convey DSM-CC download data messages). The signaling channel conveys the download control messages which are transported in DSM-CC sections with table_id equal to 0x3C (these sections convey DSM-CC download control messages).

The location of the DSM-CC Data Carousel is announced in the Program Map Table by the inclusion of an association_tag_descriptor structure defined by DSM-CC which binds the TransMux instance to a particular packet identifier (PID). The Stream Map Table is transmitted in the form of a directory service in DownloadInfoIndication() control messages. This table associates each elementary stream identifier (ES_ID) used in applications with a channelAssociationTag which uniquely identifies the channel in the underlying session. The value of the channelAssociationTag field is tied to the value of the module identifier moduleId.

Kommentar: This paragraph is fuzzy. I removed some references to DMIF, but needs reworking.

The scene description streams (BIFS, BIFS updates, BIFS animation streams) and object descriptors streams are transmitted in particular modules in the data carousel. These streams are identified in the First object descriptor stream. The remaining modules are used to transmit the various elementary streams which provide the data necessary to compose the scene.

The Figure below provides an overview of the system. The dotted lines represent the interface between the application and the underlying DMIF session (DMIF Application Interface or DAI). In the Figure below, the **TransMuxAssociationTag** abstracts the location of the MPEG-2 Transport Stream packets conveying the DSM-CC sections. Each channel in the designated TS packets is identified by a **channelAssociationTag** (CAT). The value of a CAT is built from the module identifier **moduleId** value published in the directory messages (DSM-CC downloadInfoIndication() messages) transmitted on the signaling channel. Consecutive chunks of data modules are transported via DSM-CC downloadDataBlock messages. The payload of each of these messages is viewed as a FlexMux SL packet. The SL packet header is transported in the message header preceding the downloadDataBlock message. The application requests to open a channel by means of the DA_ChannelAdd() primitive of the DAI. The **ES_id** field is an input argument which identifies the desired elementary stream in the object descriptor. The underlying DMIF session replies by providing a run-time handle to the multiplexed channel carrying the elementary stream. This handle is the **channelHandle** and is unique in the application regardless how many sessions are open. Subsequently, the DMIF session informs the application of the arrival of new data by means of the DA_Data() primitive of the DAI.

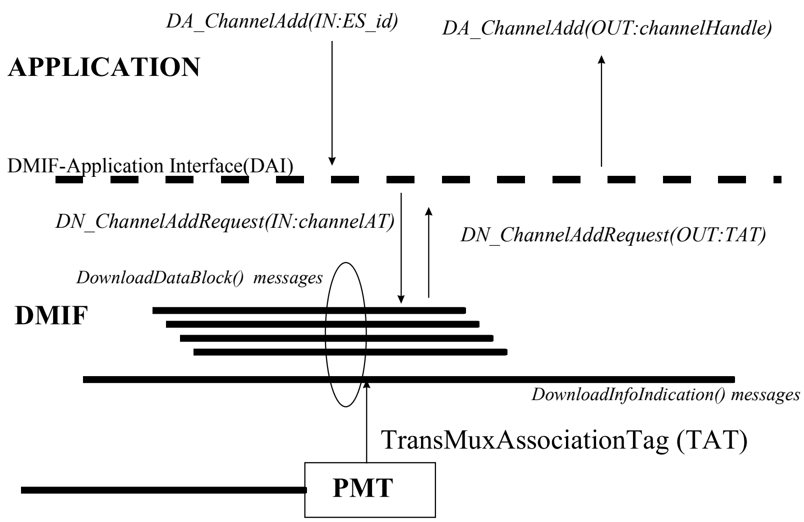


Figure C-2: Overview of ISO/IEC 13818-6 use as a FlexMux

C.2.4 ISO/IEC 13818-6 FlexMux Specification

C.2.4.1 Program Map Table

The Program Map Table (see Table 2-28 in Subclause 2.4.4.8 of ISO/IEC 13818-1) is acquired from PSI sections with **table_id** value **0x02.**

Table C-1: Transport Stream Program Map Section

Syntax	No. of bits	Mnemonic
TS_program_map_section(){		
table_id	8	uimsbf
section_syntax_indicator	1	bslbf
'0'	1	bslbf
reserved	2	bslbf
section_length	12	uimsbf
program_number	16	uimsbf
reserved	2	bslbf
version_number	5	uimsbf
current_next_indicator	1	bslbf
section_number	8	uimsbf

Syntax	No. of bits	Mnemonic
last_section_number	8	uimsbf
reserved	3	bslbf
PCR_PID	13	uimsbf
reserved	4	bslbf
program_info_length	12	uimsbf
for(I=0I<N;I++){		
descriptor()		
}		
for(I=0;I<N1;I++){		
stream_type	8	uimsbf
reserved	3	bslbf
elementary_PID	13	uimsbf
reserved	4	bslbf
ES_info_length	12	uimsbf
for(j=0;j<N2;j++){		
descriptor()		
}		
}		
CRC_32	32	rpchof
}		

The data carousel resides in the stream identified by **stream_type** value **0x0B**. Table 9-4 in section 9.2.3 of ISO/IEC 13818-6 specifies that the **stream_type** value **0x0B** signals the transmission of a DSM-CC section conveying a DSM-CC User-to-Network message. The only permitted User-to-Network messages shall be the Download protocol messages defined in Subclause 7.3 of ISO/IEC 13818-6.

With the Broadcast Data Carousel are associated two descriptors in the descriptor loop following the **ES_info_length** field. The carousel identifier descriptor features a carousel identifier field **carousel_id** which identifies the carousel at the service level. The **descriptor_tag** value for this descriptor is **0x13** (19). See Table 11-2 in Subclause 11.5.1 of ISO/IEC 13818-6 for a complete definition of this descriptor. The second descriptor is the association_tag_descriptor (see Table 11-3 in Subclause 11.5.2 of ISO/IEC 13818-6) which binds the **associationTag** with the TransMux channel which in this case is the Transport Stream packets identified by the field **elementary_PID**.

The value **PCR_PID** specifies the elementary stream carrying the Clock Reference common to all the 13818-6 FlexMux channels.

Table C-2: Association Tag Descriptor

Syntax	No. of bits	Mnemonic
association_tag_descriptor(){		
descriptor_tag	8	uimsbf
descriptor_length	8	uimsbf
association_tag	16	uimsbf
use	16	uimsbf
selector_byte_length	8	uimsbf
for(n=0; n<selector_byte_length		
{		
selector_byte	8	uimsbf
}		
for(I=0;I<N;I++){		
private_data_byte	8	uimsbf

}		
}		

The descriptor_tag value is 20 (0x14).

The field **association_tag** shall convey a copy of the **TransMuxAssociationTag** (TAT) value and the **use** field value **0x0100** shall be reserved to characterize the channel as “13818-1 TransMux Channel”.

C.2.4.2 Framing of ISO/IEC 13818-6 FlexMux

An elementary stream of **stream_type** equal to **0x0B** carries the DSM-CC Download protocol messages. In this case the Transport Stream is made of DSMCC_section() structures which convey either download control or download data messages. In applications using this Final Committee Draft of International Standard, download control messages make up the application signaling channel and download data messages form a FlexMux instance. The DSMCC_section structure is defined in Table 9-2 of Subclause 9.2.2. of ISO/IEC 13818-6, shown below.

Table C-3: DSM-CC Section

Syntax	**No. of bits**	**Mnemonic**
DSMCC_section(){		
table_id	8	uimsbf
section_syntax_indicator	1	bslbf
private_indicator	1	bslbf
reserved	2	bslbf
dsmcc_section_length	12	uimsbf
table_id_extension	16	uimsbf
reserved	2	bslbf
version_number	5	uimsbf
current_next_indicator	1	bslbf
section_number	8	uimsbf
last_section_number	8	uimsbf
if(table_id == 0x3A){		
LLCSNAP()		
}		
else if(table_id == 0x3B){		
userNetworkMessage()		
}		
else if(table_id == 0x3C){		
downloadDataMessage()		
}		
else if(table_id == 0x3D){		
DSMCC_descriptor_list()		
}		
else if(table_id == 0x3E){		
for(I=0;I<dsmcc_section_length-9;I++){		
private_data_byte	8	bslbf
}		
}		
if(section_syntax_indicator == ‘0’){		
checksum	32	uimsbf
}		
else {		
CRC_32	32	rpchof

}		

Table C-4: DSM-CC table_id Assignment

table_id	DSMCC Section Type
0x00 - 0x37	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
0x38 - 0x39	ISO/IEC 13818-6 reserved
0x3A	DSM-CC sections containing multiprotocol encapsulated data
0x3B	DSM-CC sections containing U-N Messages, except Download Data Messages.
0x3C	DSM-CC sections containing Download Data Messages
0x3D	DSM-CC sections containing Stream Descriptors
0x3E	DSM-CC sections containing private data
0x3F	ISO/IEC 13818-6 reserved
0x40 - 0xFE	User private
0xFF	forbidden

The application signaling channel is recognized by the fact that associated DSMCC_sections have a **table_id** value equal to **0x3B**. DSMCC_section() structures with **table_id** equal to **0x3C** belong to the 13818-6 FlexMux. In this case, the 13818-6 FlexMux PDU is defined to be the portion of the DSM-CC section which starts after the reserved field to the Checksum or CRC_32 field not included. This is illustrated in the figure below.

DSM-CC SECTION					
table_id	section_syntax_indicator	private_indicator	reserved	MPEG-2 FlexMux PDU	Checksum or CRC_32

Figure C-3: Overview of ISO/IEC 13818-6 FlexMux Framing

C.2.4.3 Stream Map Table

The Stream Map Table links elementary stream Identifiers (**ES_id**) used by the scene description and the application to the **channelAssociationTag** (CAT) value used by the underlying DMIF session to refer to the stream. In the case of encapsulation of ISO/IEC 14496 streams into the ISO/IEC 13818-6 download protocol, the **channelAssociationTag** shall be the 4 bytes field defined as (**TransmuxAssociationTag** << 16 + **moduleId**). The Stream Map Table shall be implicitly conveyed in downloadInfoIndication messages. DownloadInfoIndication messages are conveyed in streams of type **stream_type** equal to **0x0B** and are classified as download control messages (Subclause 7.3.2 of ISO/IEC 13818-6). DSM-CC sections carrying DownloadInfoIndication messages have therefore a **table_id** value equal to **0x3B**. DownloadInfoIndication messages provide a directory service announcing the data modules available in the carousel. Each message is preceded by a dsmccMessageHeader message header (table 2-1 in section 2 of ISO/IE 13818-6)

Table C-5: DSM-CC Message Header

Syntax	Num. Of Bits
dsmccMessageHeader(){	
protocolDiscriminator	8
dsmccType	8
messageId	16
transactionId	32
reserved	8

adaptationLength	8
messageLength	16
if(adaptationLength > 0){	
dsmccAdaptationHeader()	
}	
}	

- protocolDiscriminator shall be set to **0x11** as specified in chapter 2 of ISO/IEC 13818-6
- **dsmccType** shall be set to **0x03** to indicate that a download message follows (see table 2-2 in chapter 2 of ISO/IEC 13818-6).
- **messageId** shall be set to **0x1002** as specified in table 7-4 in section 7-3 of ISO/IEC 13818-6
- the two most significant bits of **transaction_id** are set to **0x01** to indicate that this field is assigned by the server (see table 2-3 in chapter 2 of ISO/IEC 13818-6).

Table C-6: Adaptation Header

Syntax	Num of bits
dsmccAdaptationHeader(){	
adaptationType	8
for(I=0;I<adaptationLength-1;I++){	
adaptationDataByte	8
}	
}	

Table C-7: DSM-CC Adaptation Types

Adaptation Type	Description
0x00	ISO/IEC 13818-6 reserved
0x01	DSM-CC Conditional Access adaptation format.
0x02	DSM-CC User ID adaptation format
0x03	DIImsgNumber adaptation format
0x04-0x7F	ISO/IEC 13818-6 Reserved.
0x80-0xFF	User defined adaptation type

Any downloadInfoIndication message shall include a dsmccAdaptationHeader with an **adaptationType** value equal to **0x03** to specify the DownloadInfoIndication message number.

Table C-8: DownloadInfoIndication Message

Syntax	Num. Of Bits
DownloadInfoIndication(){	
dsmccMessageHeader()	
downloadId	32
blockSize	16
windowSize	8
ackPeriod	8
tCDownloadWindow	32
tCDownloadScenario	32
compatibilityDescriptor()	
numberOfModules	16
for(I=0;I<numberOfModules;I++){	
moduleId	16

moduleSize	32
moduleVersion	8
moduleInfoLength	8
for(j=0;j<moduleInfoLength;j++){	
moduleInfoByte	8
}	
}	
privateDataLength	16
for(I=0;I<privateDataLength;I++){	
privateDataByte	8
}	
}	

- downloadId conveys a copy of carousel_id
- **windowSize** shall be set to **0x00** (section 7.3.2 of ISO/IEC 13818-6)
- **ackPeriod** shall be set to **0x00** (section 7.3.2 of ISO/IEC 13818-6)
- **tcDownloadWindow** shall be set to **0x00** (section 7.3.2 of ISO/IEC 13818-6)

The value of **moduleSize** shall always be set to **0**.The field **moduleId** shall convey a copy of the FlexMux channel number. In addition, the **moduleInfoByte** fields shall convey the **ES_id** field values used by applications to refer to the individual elementary streams. The **moduleInfoByte** fields shall follow the following format:

Table C-9: ModuleInfoBytes

moduleInfoBytes(){	Num. of Bits	Mnemonic
moduleInfoDescription	8	uimsbf
ES_id	15	uimsbf
FirstODstreamFlag	1	uimsbf
}		

The semantics of the fields are as follows:

moduleInfoDescription - This 8 bit field describes how to interpret the following moduleInfoBytes fields. The value of this field shall be set to **0xF0**.

ES_id – This 15 bit field conveys a copy of the MPEG-4 elementary stream Identifier associated the elementary stream conveyed in the FlexMux channel identified by **moduleId**. The value of **ES_id** is defined to be equal to ((**ObjectDescriptorId** << 5) | **ES_number**).

Kommentar: Refer to the appoprriate ESM subclause.

FirstODstreamFlag - This 1 bit field signals the presence of the first object descriptor stream. The value **1** indicates that the FlexMux channel identified by **moduleId** conveys the First ObjectDescriptor stream. This stream shall at least include an ES_Descriptor associated with the scene description stream and an ES_Descriptor associated with the object descriptor stream. The value **0** indicates that the FlexMux channel identified by **moduleId** does not convey the First object descriptor stream.

The Stream Map Table is implicitly defined by the DownloadInfoIndication message which allows MPEG-4 clients to resolve any **ES_id** value to a particular **channelAssociationTag** (equal to (**TransmuxAssociationTag** << 16) + **moduleId**). The value of **moduleId** is published in the DownloadInfoIndication message whereas the **TransmuxAssociationTag** is published in the association_tag_descriptor() residing in the PMT.

C.2.4.4 ISO/IEC 13818 TransMux Channel

DSM-CC sections with **table_id** value equal to **0x3C** convey DownloadDataBlock messages. Each DownloadDataBlock message carries a chunk of a particular data module. The ISO/IEC 13818 packetized stream of these DSMCC_section() structures is considered to be the TransMux channel. This TransMux channel shall be uniquely identified by the **association_tag** in an association_tag_descriptor residing in the PMT. The **association_tag** field conveys a copy of the **TransmuxAssociationTag** defined and managed by the underlying DMIF session. The protection layer provided by the TransMux is either error detection (using **checksum**) or error correction (using **CRC32**) depending on the value of **section_syntax_indicator** bit in the DSMCC sections.

C.2.4.5 ISO/IEC 13818 FlexMux Channel

A module can be acquired by filtering DSM-CC sections having **table_id** equal to **0x3C** and **table_id_extension** equal to the module identifier **moduleId**. These values correspond to DSM-CC sections conveying DownloadDataBlock() messages carrying chunks of a target module identified by **moduleId**. An MPEG-2 FlexMux channel shall be identified as the set of all DSM-CC sections conveying DownloadDataBlock() messages and sharing a given **moduleId** value. The MPEG-2 FlexMux channel number is the module identifier field called **moduleId**. DownloadDataBlock messages are classified as download data messages (See table 7-7 in section 7.3 of ISO/IEC 13818-6). Therefore, any DownloadDataBlock message is preceded by a dsmccDownloadDataHeader() message header (see Table 7-3 in Subclause 7.2.2.1 of ISO/IEC 13818-6).

C.2.4.5.1 ISO/IEC 13818 FlexMux PDU

A FlexMux PDU shall be considered to start in each DSMCC_section() from the **dsmcc_section_length** field included down to the last four bytes used for error protection not included (**CRC_32** or **checksum** field). The equivalent of the **index** and **length** fields defined in the MPEG-4 FlexMux are the **table_id_extension** (which conveys a copy of **moduleId** per IEC/ISO 13818-6 encapsulation rules) and the **dsmcc_section_length** fields respectively**.** The remaining DSMCC_section fields shall be considered part of the protection layer provided by the TransMux channel. The SL packet header shall be placed in the dsmccDownloadDataHeader() of the DownloadDataBlock message. The SL packet payload shall be placed in the payload of the DownloadDataBlock() message. See the figure below for an illustration.

<table>
<tr><td colspan="7">ISO/IEC 13818 FLEXMUX PDU</td></tr>
<tr><td colspan="3">DSM-CC section header</td><td colspan="2">dsmccDownloadDataHeader</td><td colspan="2">DownloadDataBlock</td></tr>
<tr><td>DSMCC_section_length</td><td>table_id_extension</td><td></td><td></td><td>ISO/IEC 14496
SL-PDU header</td><td></td><td>SL-PDU</td></tr>
</table>

Figure C-4: ISO/IEC 13818 FlexMux PDU

C.2.4.5.2 ISO/IEC 13818 SL Packet Header

The SL packet header is placed in the dsmccDownloadDataHeader preceding the downloadDataBlock message.

Table C-10: DSM-CC Download Data Header

Syntax	**Num. Of Bits**
dsmccDownloadDataHeader(){	
protocolDiscriminator	8
dsmccType	8
messageId	16
downloadId	32
reserved	8
adaptationLength	8

messageLength	16
if(adaptationLength > 0){	
dsmccAdaptationHeader()	
}	
}	

where

- **protocolDiscriminator** value is set to **0x11** as specified in Clause 2 of ISO/IEC 13818-6
- **dsmccType** is set to **0x03** as specified in Table 2-2 in Clause 2 of ISO/IEC 13818-6
- **messageId** is set to **0x1003** as specified in Table 7-4 in Subclause 7.3 of ISO/IEC 13818-6 if the message is a downloadDataBlock() message.
- download_id conveys a copy of carousel_id

The dsmccAdaptationHeader() is defined in Table 2-4 in Subclause 2.1 of ISO/IEC 13818-6. An adaptation header with an **adaptationType** value equal to **0x05** shall always be present to signal the presence of the SL packet header in the adaptation field. The following table shows the new **adaptationType** values.

Table C-11: DSM-CC Adaptation Types

Adaptation Type	Description
0x00	ISO/IEC 13818-6 reserved
0x01	DSM-CC Conditional Access adaptation format.
0x02	DSM-CC User ID adaptation format
0x03	DIImsgNumber adaptation format
0x04	ISO/IEC 13818-6 reserved
0x05	*MPEG 4 SL packet adaptation format*
0x04-0x7F	ISO/IEC 13818-6 Reserved.
0x80-0xFF	User defined adaptation type

The length of the dsmccAdaptationHeader() shall be a multiple of 4 bytes to respect the alignment rules specified in ISO/IEC 13818-6.

C.2.4.5.3 ISO/IEC 14496-1 SL packet

An SL packet is carried in the payload of a downloadDataBlock() message. The DownloadDataBlock message is defined as follows:

Table C-12: DSM-CC DownloadDataBlock() Message

Syntax	Num of bits
DownloadDataBlock(){	
dsmccDownloadDataHeader()	
moduleId	16
moduleVersion	8
reserved	8
blockNumber	16
for(I=0;I<N;I++){	
blockDataByte	8
}	
}	

C.2.4.5.4 ISO/IEC 14496 Access Unit

An access unit shall be reconstructed by concatenating SL packets according to the ordering provided by the field **blockNumber**. Following DSM-CC encapsulation rules, the 8 least significant bits of the **blockNumber** field (16 bits) shall be copied to the **section_number** field (8 bits) of the DSMCC_section().

C.2.4.6 Elementary Stream Interface

The following interface primitives are used between the Compression Layer and the sync layer in the application:

The client application requests a reference to a particular module (flexmux channel) by means of the following interface:

ESI.channelOpen.request(AppSessionId, ES_id,SLConfigDescriptor).

The sync layer exposes the channelHandle through the following interface:

ESI.channelOpen.confirm(channelHandle, Response)

Systems informs user that elementary stream is available on channel*:*

ESI.channelOpen..indication(appSessionId,ES_id,SLConfigDescriptor,channelHandle)

The application acknowledges with:

ESI.channelOpen.response(Response)

C.2.4.7 DMIF Application Interface

The application sends to the underlying DMIF session the request to open a new channel by mean of the following interface primitive:

DA_ChannelAdd.request(serviceSessionId,loop(ES_id))

The underlying DMIF session replies by means of the following interface primitive:

DA_ChannelAdd.confirm(loop(channelHandle, response))

The application then informs DMIF session that it is ready to receive data:

DA_ChannelReady (channelHandle))

DMIF privides application with data through the following interface primitive:

DA_DataCallBack (channelHandle, streamDataBuffer, streamDataLen,errorFlag)

C.3 ISO/IEC 14496 Content Embedded in a Single FlexMux Stream

This subclause gives an example of a minimal configuration of a system utilizing this Final Committee Draft of International Standard that is applicable if an application constrains a session to a single peer-to-peer interactive connection or to the access to a single stored data stream. This configuration is not intended for random access nor is it resilient to errors in the transmitted or stored data stream. Despite these limitation, this configuration has some applications, e.g., storage of data complying to this Final Committee Draft of International Standard on disk for interchange, non-real-time transmission and even real-time transmission and playback, as long as the missing random access is tolerable. For this minimal configuration, the tools defined within this specification already constitute a near complete Systems layer.

This minimal configuration consists of a FlexMux Stream. Any number of up to 256 elementary streams can be multiplexed into a Single FlexMux Stream (SFS), that offers 256 transport channels, termed FlexMux Channels.

In addition to the raw FlexMux Stream that constitutes a fully compliant FlexMux Stream according to this specification, it is necessary to specify conventions how to know what is the content of this Single FlexMux Stream. Such conventions are specified in this subclause.

C.3.1 Object Descriptor

ISO/IEC 14496 content in a Single FlexMux Stream shall always start with the object descriptor for the content carried in this SFS. This object descriptor is neither packaged in an OD message nor in an SL packet but conveyed in its raw format.

This object descriptor follows the constraints specified in Subclause 8.3.2 if the SFS contains more than one elementary stream. If the SFS contains only one elementary stream the object descriptor shall contain only one ES_Descriptor describing the properties of this elementary stream.

C.3.2 Stream Map Table

Elementary streams with arbitrary ES_IDs may be stored in the Single FlexMux Stream. In order to associate ES_IDs to the FlexMux Channels that carry the corresponding elementary streams, a Stream Map Table (SMT) is required.

The Stream Map Table shall immediately follow the initial object descriptor in the SFS. The syntax and semantics of the SMT is specified here.

C.3.2.1 Syntax

```
class StreamMapTable {
   bit(8) streamCount;
   for (i=0; i<streamCount; i++) {
      bit(16) ES_Id;
      bit(8) FlexMuxChannel;
   }
}
```

C.3.2.2 Semantics

`streamCount` is the number of streams for which the stream association is conveyed in this table.

`ES_ID` is a unique label for an elementary stream within its name scope as defined in Subclause 8.3.3.

`FlexMuxChannel` is the FlexMux Channel number within the current FlexMux Stream.

C.3.3 Single FlexMux Stream Payload

The remainder of the Single FlexMux Stream, following the first object descriptor and the Stream Map Table shall consist of FlexMux packets that encapsulate the SL packets of one or more SL-packetized streams.

The configuration of the SL packet headers of the individual SL-packetized streams is known from their related ES_Descriptors that are conveyed either in the first object descriptor or in additional object descriptors that are conveyed in an ObjectDescriptorStream that has been established by means of the first object descriptor.

Annex D: View Dependent Object Scalability (Normative)

D.1 Introduction

Coding of View-Dependent Scalability (VDS) parameters for texture can provide for efficient incremental decoding of 3D images (e.g. 2D texture mapped onto a 3D mesh such as terrain). Corresponding tools from the Visual and Systems parts of this specification (Parts 2 and 3 respectively) are used in conjunction with downstream and upstream channels of a decoding terminal. The combined capabilities provide the means for an encoder to react to a stream of viewpoint information received from a terminal. The encoder transmits a series of coded textures optimized for the viewing conditions, which can be applied to the rendering of, textured 3D meshes by the receiving terminal. Each encoded view-dependent texture (initial texture and incremental updates) typically corresponds to a specific 3D view in the user's viewpoint that is first transmitted from the receiving terminal.

A Systems tool transmits 3D viewpoint parameters in the upstream channel back to the encoder. The encoder's response is a frequency-selective, view-dependent update of DCT coefficients for the 2D texture (based upon view-dependent projection of the 2D texture in 3D) back to the receiving terminal, along the downstream channel, for decoding by a Visual DCT tool at the receiving terminal. This bilateral communication supports interactive server-based refinement of texture for low-bandwidth transmissions to a decoding terminal that renders the texture in 3D for a user controlling the viewpoint movement. A gain in texture transmission efficiency is traded for longer closed-loop latency in the rendering of the textures in 3D. The terminal coordinates inbound texture updates with local 3D renderings, accounting for network delays so that texture cached in the terminal matches each rendered 3D view.

A method to obtain an optimal coding of 3D data is to take into account the viewing position in order to transmit only the most visible information. This approach reduces greatly the transmission delay, in comparison to transmitting all scene texture that might be viewable in 3D from the encoding database server to the decoder. At a given time, only the most important information is sent, depending on object geometry and viewpoint displacement. This technique allows the data to be streamed across a network, given that a upstream channel is available for sending the new viewing conditions to the remote database. This principle is applied to the texture data to be mapped on a 3D grid mesh. The mesh is first downloaded into the memory of the decoder using the appropriate BIFS node, and then the DCT coefficients of the texture image are updated by taking into account the viewing parameters, i.e. the field of view, the distance and the direction to the viewpoint.

D.2 Bitstream Syntax

This subclause details the bitstream syntax for the upstream data and details the rules that govern the way in which higher level syntactic elements may be combined together to generate a compliant bitstream that can be decoded correctly by the receiver.

Subclause D.2.1 is concerned with the bitstream syntax for a View Dependent Object which initializes the session at the upstream data decoder. Subclause D.2.2 is concerned with the View Dependent Object Layer and contains the viewpoint information that is to be communicated back to the encoder.

D.2.1 View Dependent Object

ViewDependentObject() {	No. of bits	Mnemonic
view_dep_object_start_code	32	bslbf
field_of_view	16	uimsbf
marker_bit	1	bslbf
xsize_of_rendering_window	16	uimsbf
marker_bit	1	bslbf
ysize_of_rendering_window	16	uimlbf
marker_bit	1	bslbf

`  do {`		
`    ViewDependentObjectLayer()`		
`  } while(nextbits_bytealigned()== view_dep_object_layer_start_code)`		
`    next_start_code()`		
`}`		

D.2.2 View Dependent Object Layer

`ViewDependentObjectLayer() {`	**No. of bits**	**Mnemonic**
view_dep_object_layer_start_code	32	bslbf
xpos1	16	uimsbf
marker_bit	1	bslbf
xpos2	16	uimsbf
marker_bit	1	bslbf
ypos1	16	uimsbf
marker_bit	1	bslbf
ypos2	16	uimsbf
marker_bit	1	bslbf
zpos1	16	uimsbf
marker_bit	1	bslbf
zpos2	16	uimsbf
marker_bit	1	bslbf
xaim1	16	uimsbf
marker_bit	1	bslbf
xaim2	16	uimsbf
marker_bit	1	bslbf
yaim1	16	uimsbf
marker_bit	1	bslbf
yaim2	16	uimsbf
marker_bit	1	bslbf
zaim1	16	uimsbf
marker_bit	1	bslbf
zaim2	16	uimsbf
`}`		

D.3 Bitstream Semantics

D.3.1 View Dependent Object

view_dep_object_start_code – The view_dep_object_start_code is the string '000001BF' in hexadecimal. It initiates a view dependent object session.

field_of_view – This is a 16-bit unsigned integer that specifies the field of view.

marker bit – This is a one bit field, set to '1', to prevent start code emulation within the bitstream.

xsize_of_rendering_window – This is a 16-bit unsigned integer that specifies the horizontal size of the rendering window.

ysize_of_rendering_window –This is a 16 unsigned integer that specifies the vertical size of the rendering window.

D.3.2 View Dependent Object Layer

view_dep_object_layer_start_code –The view_dep_object_layer_start_code is the bit string '000001BE' in hexadecimal. It initiates a view dependent object layer.

xpos1 – This is a 16 bit codeword which forms the lower 16 bit of the 32 bit integer xpos. The integer xpos is to be computed as follows: xpos = xpos1 + (xpos2 << 16). The quantities xpos, ypos, zpos describe the 3D coordinates of the viewer's position.

xpos2 – This is a 16 bit codeword which forms the upper 16 bit word of the 32 bit integer xpos.

ypos1 – This is a 16 bit codeword which forms the lower 16 bit word of the 32 bit integer ypos. The integer ypos can be computed as follows: ypos = ypos1 + (ypos2 << 16).

ypos2 – This is a 16 bit codeword which forms the upper 16 bit word of the 32 bit integer xpos.

zpos1 – This is a 16 bit codeword which forms the lower 16 bit of the 32 bit integer xpos. The integer zpos can be computed as follows: zpos = zpos1 + (zpos2 << 16).

zpos2 – This is a 16 bit codeword which forms the upper 16 bit of the 32 bit integer xpos.

xaim1 – This is a 16 bit codeword which forms the lower 16 bit of the 32 bit integer xaim. The integer xaim can be computed as follows: xaim = xaim1 + (xaim2 << 16). The quantities xaim, yaim, zaim describe the 3D position of the aim point.

xaim2 – This is a 16 bit codeword which forms the upper 16 bit of the 32 bit integer xaim.

yaim1 – This is a 16 bit codeword which forms the lower 16 bit of the 32 bit integer yaim. The integer yaim can be computed as follows: yaim = yaim1 + (yaim2 << 16).

yaim2 – This is a 16 bit codeword which forms the upper 16 bit of the 32 bit integer yaim.

zaim1 – This is a 16 bit codeword which forms the lower 16 bit of the 32 bit integer zaim. The integer zaim can be computed as follows: zaim = zaim1 + (zaim2 << 16).

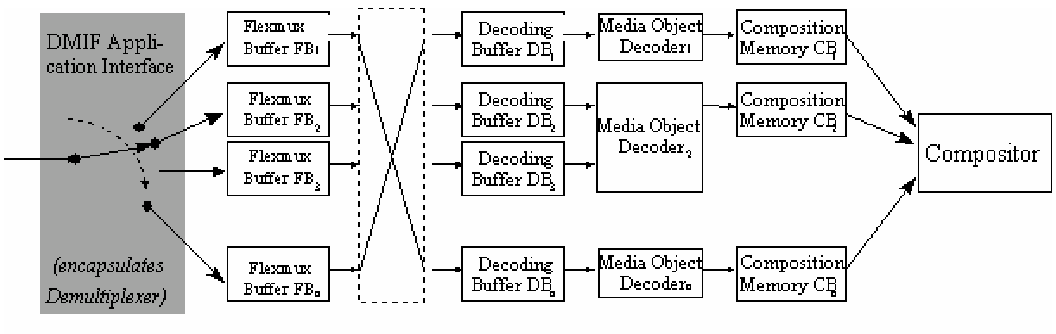
zaim2 – This is a 16 bit codeword which forms the upper 16 bit of the 32 bit integer zaim.

Annex E: System Decoder Model For FlexMux Tool (Informative)

E.1 Introduction

The different semantics of both FlexMux layer and sync layer specifications and the constraints on these semantics require exact definition of byte arrival, decoding events, composition events and the times at which these events occur.

The System Decoder Model (SDM) is a conceptual model which can be used to precisely define those terms and to model the FlexMux layer, sync layer and decoding processes in the terminal.



E.2 Definitions

E.2.1 FlexMux Streams

FlexMux layer multiplexing is a packetization performed over access unit streams, before accessing storage equipment or a network. Each FlexMux stream is usually assigned one quality of service provided by TransMux entities (outside the scope of this specification). The interface with the TransMux instance relies on the DMIF application Interface (DAI) provided by DMIF (see Part 6 of this Final Committee Draft of International Standard). One FlexMux stream corresponds to one FlexMux channel. The FlexMux syntax allows to multiplex together different access unit streams with independent OTBs.

E.2.2 FlexMux Buffer (FB)

The FlexMux buffer is a receiver buffer that contains FlexMux streams. The Systems Buffering Model enables the sender to monitor the FlexMux buffer resources that are used during a session.

E.3 Timing Model Specification

E.3.1 Access Unit Stream Bitrate

This is the bitrate between the FlexMux buffer and the decoding buffer. It can be assessed during session establishment or be conveyed in the instantBitrate field conveyed jointly with the OCR. OTB recovery also requires the acquisition of successive OCR samples. InstantBitrate fields are placed in the SL packet header as described in Subclause 10.2.3.

E.3.2 Adjusting the Receiver's OTB

For each stream that conveys OCR, it is possible for the receiver to adjust the local OTB to the encoder OTB. This can be done, for example, by well-known PLL techniques. The notion of time for each object can therefore be recovered at the receiver side.

In OCR fields, timing information is coded as the sample value of an Object Time Base. The OCR fields are carried in the SL packet headers of the SL packets for the elementary streams that have been declared at the session establishment to be carrying OCR fields. The OCR sample accuracy has to be known.

Decoders may reconstruct a replica of the OTB from these values, their respective arrival time and the SL-packetized stream's piecewise constant bitrate.

The OTB_resolution is indicated in Hz, and should be known (within a range of values), with a drift (rate of change) of the OTB_resolution expressed in Hz/s.

E.4 Buffer and Timing Model Specification

E.4.1 Elementary Decoder Model

The following simplified model is assumed for the purpose of specifying the buffer and timing models. Each elementary stream is regarded separately.

.

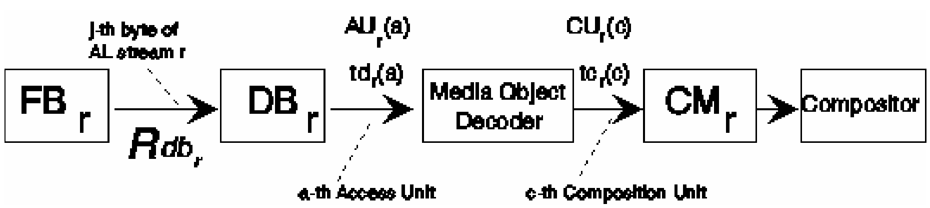


Figure E-1: Flow diagram for the system decoder model

E.4.2 Definitions

r	is an index to the different elementary streams.
FB_r	is the FlexMux layer buffer related to elementary stream r.
FBS_r	is the size of the FlexMux layer buffer related to elementary stream r. This size is measured in bytes.
Rdb_r	is the bitrate at which data enter the DB_r buffer.
j	is an index to bytes in the sync layer streams. j' and j'' can also be used.
$ta_r(j)$	indicates the time, measured in seconds, at which the j^{th} byte of the sync layer stream 'r' enters the Decoding buffer DB_r.
$AU_r(a)$	is the a^{th} access unit in elementary stream r. $AU_r(a)$ is indexed in decoding order.

E.4.3 Assumptions

E.4.3.1 Input to Decoding Buffers

Data from the sync layer stream 'r' enter DB_r buffers at a piecewise constant bitrate.

The j^{th} access unit layer byte enters at time $ta_r(j)$.

The time at which this byte enters the DB_r buffer can be recovered from the SL stream 'r' by decoding the last input OCR field encoded in the SL packet header, and by counting the bytes in the SL stream 'r'.

E.4.3.2 Buffering

Complete FlexMux stream packets which contain data from elementary stream 'r' are passed to the FlexMux buffer for stream 'r' : FB_r. This includes duplicate FlexMux packets.

Transfer of bytes between the input and the FB_r buffer is instantaneous, so that no delay is introduced.

All the bytes which enter FB_r are removed.

Bytes which are part of the SL packet are delivered to the DB_r buffer at a piecewise constant bitrate Rdb_r. Other bytes are not delivered, they are removed instantaneously.

The buffer FB_r can be emptied according to two possibilities :

a) when OCR are applied :

the Rdb_r bitrate is equal to the value indicated in the instantBitrate field present in the SL packet header or was declared at the session establishment.

Rdb_r can never be zero.

The FB_r buffer is not allowed to underflow or be empty.

b) when no OCR are applied :

When there is information in the FB_r buffer, the Rdb_r bitrate is equal to the value indicated in the instantBitrate field present in the SL packet header or was declared at the session establishment.

Rdb_r can be zero when FB_r is empty.

The FB_r buffer is allowed to underflow or be empty.

When the FB_r buffer is empty the Rdb_r bitrate is equal to zero.

Rdb_r is measured with respect to the OTB_resolution.

All the needed buffer sizes are known by the sender and conveyed to the receiver as specified in Subclause 8.3.4.

E.4.3.3 Bufer Management

FlexMux streams and FB buffers shall be constructed and sized so that the following conditions are satisfied :

- FB_r buffers shall not underflow, when OTB recovery is done using OCRs.
- FB_r, DB_r buffers shall not overflow.

Annex F: Registration Procedure (Informative)

F.1 Procedure for the request of a RID

Requesters of a RID shall apply to the Registration Authority. Registration forms shall be available from the Registration Authority. The requester shall provide the information specified in Subclause F.4. Companies and organizations are eligible to apply.

F.2 Responsibilities of the Registration Authority

The primary responsibilities of the Registration Authority administrating the registration of private data format_identifiers are outlined in this annex; certain other responsibilities maybe found in the JTC 1 Directives. The Registration Authority shall:

a) implement a registration procedure for application for a unique RID in accordance with the JTC 1 Directives;

b) receive and process the applications for allocation of an identifier from application providers;

c) ascertain which applications received are in accordance with this registration procedure, and to inform the requester within 30 days of receipt of the application of their assigned RID;

d) inform application providers whose request is denied in writing with 30 days of receipt of the application, and to consider resubmissions of the application in a timely manner;

e) maintain an accurate register of the allocated identifiers. Revisions to format specifications shall be accepted and maintained by the Registration Authority;

f) make the contents of this register available upon request to National Bodies of JTC 1 that are members of ISO or IEC, to liaison organizations of ISO or IEC and to any interested party;

g) maintain a data base of RID request forms, granted and denied. Parties seeking technical information on the format of private data which has a RID shall have access to such information which is part of the data base maintained by the Registration Authority.;

h) report its activities annually to JTC 1, the ITTF, and the SC 29 Secretariat, or their respective designees; and

i) accommodate the use of existing RIDs whenever possible.

F.3 Contact information for the Registration Authority

To Be Determined

F.4 Responsibilities of Parties Requesting a RID

The party requesting a format_identifier shall:

a) apply using the Form and procedures supplied by the Registration Authority;

b) include a description of the purpose of the registered bitstream, and the required technical details as specified in the application form;

c) provide contact information describing how a complete description can be obtained on a non-discriminatory basis;

d) agree to institute the intended use of the granted RID within a reasonable time frame; and

e) to maintain a permanent record of the application form and the notification received from the Registration Authority of a granted RID.

F.5 Appeal Procedure for Denied Applications

The Registration Management Group is formed to have jurisdiction over appeals to denied request for a RID. The RMG shall have a membership who is nominated by P- and L-members of the ISO technical committee responsible for this specification. It shall have a convenor and secretariat nominated from its members. The Registration Authority is entitled to nominate one non-voting observing member.

The responsibilities of the RMG shall be:

a) to review and act on all appeals within a reasonable time frame;

b) to inform, in writing, organizations which make an appeal for reconsideration of its petition of the RMGs disposition of the matter;

c) to review the annual report of the Registration Authorities summary of activities; and

d) to supply Member Bodies of ISO and National Committees of IEC with information concerning the scope of operation of the Registration Authority.

F.6 Registration Application Form

F.6.1 Contact Information of organization requesting a RID

Organization Name:
Address:

Telephone:
Fax:
E-mail:
Telex:

F.6.2 Request for a specific RID

Note: If the system has already been implemented and is in use, fill in this item and item F.6.3 and skip to F.6.6, otherwise leave this space blank and skip to F.6.4)

F.6.3 Short description of RID that is in use and date system was implemented

F.6.4 Statement of an intention to apply the assigned RID

F.6.5 Date of intended implementation of the RID

F.6.6 Authorized representative

Name:
Title:
Address:
Email:

Signature __________________________________

F.6.7 For official use of the Registration Authority

Registration Rejected _____

Reason for rejection of the application:

Registration Granted ______ Registration Value ____________________

Attachment 1: Attachment of technical details of the registered data format.

Attachment 2: Attachment of notification of appeal procedure for rejected applications.

Annex G: The QoS Management Model for ISO/IEC 14496 Content (Informative)

Kommentar: Fix smart quotes (how did they ever get screwed???)

The Quality of Service aspects deserve particular attention in this specification: the ability of the standard to adapt to different service scenarios is affected by its ability to consistently manage Quality of Service requirements. Current techniques on error resilience are already effective, but are not and will not be able to satisfy every possible requirement.

In general terms, the end-user acceptance of a particular service varies depending on the kind of service. As an example, person to person communication is severely affected by the audio quality, while it can tolerate variations in the video quality. However, television broadcast with higher video and lower audio quality may be acceptable depending on the program being transmitted. The acceptability of a particular service thus depends very much on the service itself. It is not possible to define universal Quality of Service levels that are correct in all circumstances. Thus the most suitable solution is to let the content creator decide what QoS the end-user should obtain for every particular elementary stream: the author has the best knowledge of the service.

The QoS so defined represents the QoS that should be offered to the end-user, i.e., the QoS at the output of the receiving equipment. This may be the output of the decoder, but may also take into account the compositor and renderer if they significantly impact on the QoS of the stream as seen by the end-user, and if a capacity for processing a specific stream can be quantified. Note that QoS information is not mandatory. In the absence of QoS requirements, a best effort approach should be pursued. This QoS concept is defined as *total QoS*.

In this specification the information concerning the total QoS of a particular Elementary Stream is carried in a QoS Descriptor as part of its Elementary Stream Descriptor (ES_Descriptor). The receiving terminal, upon reception of the ES_Descriptor, is therefore aware of the characteristics of the Elementary Stream and of the total QoS to be offered to the end-user. Moreover the receiving terminal knows about its own performance capabilities. It is therefore the only possible entity able to compute the Quality of Service to be requested to the Delivery Layer in order to fit the user requirements. Note that this computation could also ignore/override the total QoS parameters.

The QoS that is requested to the Delivery Layer is named *media QoS*, since it is expressed with a semantic which is media oriented. The Delivery Layer will process the requests, determine whether to bundle multiple Elementary Streams into a single network connection (TransMux) and compute the QoS for the network connection, using the QoS parameters as defined by the network infrastructure. This QoS concept is named *network QoS*, since it is specific for a particular network technology.

The above categorization of the various QoS concepts managed in this specification may suggest that this issue is only relevant when operating in a network environment. However the concepts are of general value, and are applicable to systems operating on local files as well, when taking into account the overall capacity of the system.

Annex H: Node Coding Parameters

H.1 Node Coding Tables

The Node Coding Tables contain the following information:

- Name of the node.
- List of Node Data Types (NDTs) to which the node belongs.
- List of corresponding nodeIDs.
- For each field:
- – The DEF, IN, OUT and/or DYN field IDs
- – The minimum and maximum values of the field (used for quantization)
- – The quantization category.
- – The animation category.

The table template is as follows:

Node Coding Table

Node Name	**Node Data Type list**					**list id/NDT**		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[min, max]***	***Quantization***	***Animation***

H.1.1 AnimationStream

AnimationStream	SFWorldNode SF3DNode SF2DNode SFStreamingNode					0000001 000001 00001 001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
loop	SFBool	000	000	000				
speed	SFFloat	001	001	001		[-I, +I]	0	7
startTime	SFTime	010	010	010		[-I, +I]		
stopTime	SFTime	011	011	011		[-I, +I]		
url	MFURL	100	100	100				
isActive	SFBool			101				

H.1.2 Appearance

Appearance	SFWorldNode SFAppearanceNode					0000010 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
material	SFMaterialNode	00	00	00				
texture	SFTextureNode	01	01	01				
textureTransform	SFTextureTransform	10	10	10				

	Node							

H.1.3 AudioClip

AudioClip	SFWorldNode SFAudioNode SFStreamingNode					0000011 001 010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
description	SFString	000	000	000				
loop	SFBool	001	001	001				
pitch	SFFloat	010	010	010		[0, +I]	0	7
startTime	SFTime	011	011	011		[-I, +I]		
stopTime	SFTime	100	100	100		[-I, +I]		
url	MFURL	101	101	101				
duration_changed	SFTime			110				
isActive	SFBool			111				

H.1.4 AudioDelay

AudioDelay	SFWorldNode SFAudioNode					0000100 010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MFAudioNode		00					
removeChildren	MFAudioNode		01					
children	MFAudioNode	00	10	0				
delay	SFTime	01	11	1		[0, +I]		
numChan	SFInt32	10				[0, 255]	13 8	
phaseGroup	MFInt32	11				[0, 255]	13 8	

H.1.5 AudioFX

AudioFX	SFWorldNode SFAudioNode					0000101 011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MFAudioNode		000					
removeChildren	MFAudioNode		001					
children	MFAudioNode	000	010	00				
orch	SFBuffer	001	011	01				
score	SFBuffer	010	100	10				
params	MFFloat	011	101	11		[-I, +I]	0	7
numChan	SFInt32	100				[0, 255]	13 8	
phaseGroup	MFInt32	101				[0, 255]	13 8	

H.1.6 AudioMix

AudioMix	SFWorldNode SFAudioNode					0000110 100		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
addChildren	MFAudioNode		000					
removeChildren	MFAudioNode		001					
children	MFAudioNode	000	010	00				
numInputs	SFInt32	001	011	01		[1, 255]	13 8	
matrix	MFFloat	010	100	10		[0, 1]	0	7
numChan	SFInt32	011				[0, 255]	13 8	
phaseGroup	MFInt32	100				[0, 255]	13 8	

H.1.7 AudioSource

AudioSource	SFWorldNode SFAudioNode SFStreamingNode					0000111 101 011		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
url	MFURL	000	00	00				
pitch	SFFloat	001	01	01		[0, +I]	0	7
startTime	SFTime	010	10	10				
stopTime	SFTime	011	11	11				
numChan	SFInt32	100				[1, 255]	13 8	
phaseGroup	MFInt32	101				[1, 255]	13 8	

H.1.8 AudioSwitch

AudioSwitch	SFWorldNode SFAudioNode					0001000 110		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
addChildren	MFAudioNode		00					
removeChildren	MFAudioNode		01					
children	MFAudioNode	00	10	0				
whichChoice	MFInt32	01	11	1				
numChan	SFInt32	10				[0, 255]	13 8	
phaseGroup	MFInt32	11				[0, 255]	13 8	

H.1.9 Background

Background	SFWorldNode SF3DNode					0001001 000010		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
set_bind	SFBool		0000					
groundAngle	MFFloat	0000	0001	0000	00	[0, 1.5707963]	6	3

groundColor	MFColor	0001	0010	0001	01	[0, 1]	4	2
backURL	MFURL	0010	0011	0010				
frontURL	MFURL	0011	0100	0011				
leftURL	MFURL	0100	0101	0100				
rightURL	MFURL	0101	0110	0101				
topURL	MFURL	0110	0111	0110				
skyAngle	MFFloat	0111	1000	0111	10	[0, 6.2831853]	6	3
skyColor	MFColor	1000	1001	1000	11	[0, 1]	4	2
isBound	SFBool			1001				

H.1.10 Background2D

Background2D	SFWorldNode SF2DNode					0001010 00010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_bind	SFBool		0					
url	MFURL		1	0				
isBound	SFBool			1				

H.1.11 Billboard

Billboard	SFWorldNode SF3DNode					0001011 000011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF3DNode		00					
removeChildren	MF3DNode		01					
children	MF3DNode	00	10	0				
axisOfRotation	SFVec3f	01	11	1			9	
bboxCenter	SFVec3f	10				[-I, +I]	1	
bboxSize	SFVec3f	11				[0, +I]	11	

H.1.12 Body

Body	SFWorldNode SFBodyNode					0001100 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
renderedBodyPlaceHolder	MF3DNode							

H.1.13 Box

Box	SFWorldNode SFGeometryNode					0001101 00001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***

size	SFVec3f					[0, +I]	11	0

H.1.14 Circle

Circle	SFWorldNode SFGeometryNode					0001110 00010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
radius	SFFloat					[0, +I]	11	7

H.1.15 Collision

Collision	SFWorldNode SF3DNode					0001111 000100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF3DNode		00					
removeChildren	MF3DNode		01					
children	MF3DNode	000	10	00				
collide	SFBool	001	11	01				
bboxCenter	SFVec3f	010				[-I, +I]	1	
bboxSize	SFVec3f	011				[0, +I]	11	
proxy	SF3DNode	100						
collideTime	SFTime			10				

H.1.16 Color

Color	SFWorldNode SFColorNode					0010000 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
color	MFColor					[0, 1]	4	2

H.1.17 ColorInterpolator

ColorInterpolator	SFWorldNode SF3DNode SF2DNode					0010001 000101 00011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFColor	1	10	01		[0, 1]	4	
value_changed	SFColor			10				

H.1.18 Composite2DTexture

Composite2DTexture	SFWorldNode SFTextureNode	0010010 001

Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
addChildren	MF2DNode		000					
removeChildren	MF2DNode		001					
children	MF2DNode	00	010	00				
pixelWidth	SFInt32	01	011	01		[0, 65535]	13 16	
pixelHeight	SFInt32	10	100	10		[0, 65535]	13 16	

H.1.19 Composite3DTexture

Composite3DTexture	SFWorldNode SFTextureNode					0010011 010		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
addChildren	MF3DNode		0000					
removeChildren	MF3DNode		0001					
children	MF3DNode	000	0010	000				
pixelWidth	SFInt32	001	0011	001		[0, 65535]	13 16	
pixelHeight	SFInt32	010	0100	010		[0, 65535]	13 16	
background	SF3DNode	011	0101	011				
fog	SF3DNode	100	0110	100				
navigationInfo	SF3DNode	101	0111	101				
Viewpoint	SF3DNode	110	1000	110				

H.1.20 CompositeMap

CompositeMap	SFWorldNode SF3DNode					0010100 000110		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
addChildren	MF2DNode		00					
removeChildren	MF2DNode		01					
children	MF2DNode	0	10	0				
sceneSize	SFVec2f	1	11	1		[-I, +I]	2	1

H.1.21 Conditional

Conditional	SFWorldNode SF3DNode SF2DNode					0010101 000111 00100		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
activate	SFBool		00					
reverseActivate	SFBool		01					
buffer	SFBuffer		10	0				
isActive	SFBool			1				

H.1.22 Cone

Cone	SFWorldNode SFGeometryNode					0010110 00011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
bottomRadius	SFFloat	00			0	[0, +I]	11	7
height	SFFloat	01			1	[0, +I]	11	7
side	SFBool	10						
bottom	SFBool	11						

H.1.23 Coordinate

Coordinate	SFWorldNode SFCoordinateNode					0010111 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
point	MFVec3f					[-I, +I]	1	0

H.1.24 Coordinate2D

Coordinate2D	SFWorldNode SFCoordinate2DNode					0011000 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
point	MFVec2f					[-I, +I]	2	1

H.1.25 CoordinateInterpolator

CoordinateInterpolator	SFWorldNode SF3DNode					0011001 001000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFVec3f	1	10	01		[-I, +I]	1	
value_changed	MFVec3f			10				

H.1.26 Curve2D

Curve2D	SFWorldNode SFGeometryNode					0011010 00100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
points	SFCoordinate2DNode	0	0	0				
fineness	SFInt32	1	1	1		[-I, +I]		

H.1.27 Cylinder

Cylinder	SFWorldNode SFGeometryNode					0011011 00101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
bottom	SFBool	000						
height	SFFloat	001			0	[0, +I]	11	7
radius	SFFloat	010			1	[0, +I]	11	7
side	SFBool	011						
top	SFBool	100						

H.1.28 DirectionalLight

DirectionalLight	SFWorldNode SF3DNode					0011100 001001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
ambientIntensity	SFFloat	000	000	000	00	[0, 1]	4	7
color	SFColor	001	001	001	01	[0, 1]	4	2
direction	SFVec3f	010	010	010	10		9	4
intensity	SFFloat	011	011	011	11	[0, 1]	4	7
on	SFBool	100	100	100				

H.1.29 DiscSensor

DiscSensor	SFWorldNode SF2DNode					0011101 00101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
autoOffset	SFBool	000	000	0000				
center	SFVec2f	001	001	0001		[-I, +I]	1	
enabled	SFBool	010	010	0010				
maxAngle	SFFloat	011	011	0011		[-6.2831853, 6.2831853]	6	
minAngle	SFFloat	100	100	0100		[-6.2831853, 6.2831853]	6	
offset	SFFloat	101	101	0101		[-I, +I]	6	
isActive	SFBool			0110				
rotation_changed	SFFloat			0111				
trackPoint_changed	SFVec2f			1000				

H.1.30 ElevationGrid

ElevationGrid	SFWorldNode SFGeometryNode					0011110 00110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_height	MFFloat		00					
color	SFColorNode	0000	01	00				

normal	SFNormalNode	0001	10	01				
texCoord	SFTextureCoordinateNode	0010	11	10				
height	MFFloat	0011				[-I, +I]	11	7
ccw	SFBool	0100						
colorPerVertex	SFBool	0101						
creaseAngle	SFFloat	0110				[0, 6.2831853]	6	
normalPerVertex	SFBool	0111						
solid	SFBool	1000						
xDimension	SFInt32	1001				[0, +I]	11	
xSpacing	SFFloat	1010				[0, +I]	11	
zDimension	SFInt32	1011				[0, +I]	11	
zSpacing	SFFloat	1100				[0, +I]	11	

H.1.31 Expression

Expression	SFWorldNode SFExpressionNode					0011111 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
expression_select1	SFInt32	000	000	000		[0, 31]	13 5	
expression_intensity1	SFInt32	001	001	001		[0, 63]	13 6	
expression_select2	SFInt32	010	010	010		[0, 31]	13 5	
expression_intensity2	SFInt32	011	011	011		[0, 63]	13 6	
init_face	SFBool	100	100	100				
expression_def	SFBool	101	101	101				

H.1.32 Extrusion

Extrusion	SFWorldNode SFGeometryNode					0100000 00111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_crossSection	MFVec2f		00					
set_orientation	MFRotation		01					
set_scale	MFVec2f		10					
set_spine	MFVec3f		11					
beginCap	SFBool	0000						
ccw	SFBool	0001						
convex	SFBool	0010						
creaseAngle	SFFloat	0011				[0, 6.2831853]	6	
crossSection	MFVec2f	0100			00	[-I, +I]	2	1
endCap	SFBool	0101						
orientation	MFRotation	0110			01	[-I, +I]	10	6

scale	MFVec2f	0111			10	[0, +I]	7	1
solid	SFBool	1000						
spine	MFVec3f	1001			11	[-I, +I]	1	0

H.1.33 FAP

FAP	SFWorldNode SFFAPNode					01000011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
viseme	SFVisemeNode	0000000	0000000	0000000				
expression	SFExpressionNode	0000001	0000001	0000001				
open_jaw	SFInt32	0000010	0000010	0000010		[0, +I]	0	
lower_t_midlip	SFInt32	0000011	0000011	0000011		[-I, +I]	0	
raise_b_midlip	SFInt32	0000100	0000100	0000100		[-I, +I]	0	
stretch_l_corner	SFInt32	0000101	0000101	0000101		[-I, +I]	0	
stretch_r_corner	SFInt32	0000110	0000110	0000110		[-I, +I]	0	
lower_t_lip_lm	SFInt32	0000111	0000111	0000111		[-I, +I]	0	
lower_t_lip_rm	SFInt32	0001000	0001000	0001000		[-I, +I]	0	
lower_b_lip_lm	SFInt32	0001001	0001001	0001001		[-I, +I]	0	
lower_b_lip_rm	SFInt32	0001010	0001010	0001010		[-I, +I]	0	
raise_l_cornerlip	SFInt32	0001011	0001011	0001011		[-I, +I]	0	
raise_r_cornerlip	SFInt32	0001100	0001100	0001100		[-I, +I]	0	
thrust_jaw	SFInt32	0001101	0001101	0001101		[0, +I]	0	
shift_jaw	SFInt32	0001110	0001110	0001110		[-I, +I]	0	
push_b_lip	SFInt32	0001111	0001111	0001111		[-I, +I]	0	
push_t_lip	SFInt32	0010000	0010000	0010000		[-I, +I]	0	
depress_chin	SFInt32	0010001	0010001	0010001		[0, +I]	0	
close_t_l_eyelid	SFInt32	0010010	0010010	0010010		[-I, +I]	0	
close_t_r_eyelid	SFInt32	001001	001001	001001		[-I, +I]	0	

		1	1	1				
close_b_l_eyelid	SFInt32	0010100	0010100	0010100		[-I, +I]	0	
close_b_r_eyelid	SFInt32	0010101	0010101	0010101		[-I, +I]	0	
yaw_l_eyeball	SFInt32	0010110	0010110	0010110		[-I, +I]	0	
yaw_r_eyeball	SFInt32	0010111	0010111	0010111		[-I, +I]	0	
pitch_l_eyeball	SFInt32	0011000	0011000	0011000		[-I, +I]	0	
pitch_r_eyeball	SFInt32	0011001	0011001	0011001		[-I, +I]	0	
thrust_l_eyeball	SFInt32	0011010	0011010	0011010		[-I, +I]	0	
thrust_r_eyeball	SFInt32	0011011	0011011	0011011		[-I, +I]	0	
dilate_l_pupil	SFInt32	0011100	0011100	0011100		[0, +I]	0	
dilate_r_pupil	SFInt32	0011101	0011101	0011101		[0, +I]	0	
raise_l_i_eyebrow	SFInt32	0011110	0011110	0011110		[-I, +I]	0	
raise_r_i_eyebrow	SFInt32	0011111	0011111	0011111		[-I, +I]	0	
raise_l_m_eyebrow	SFInt32	0100000	0100000	0100000		[-I, +I]	0	
raise_r_m_eyebrow	SFInt32	0100001	0100001	0100001		[-I, +I]	0	
raise_l_o_eyebrow	SFInt32	0100010	0100010	0100010		[-I, +I]	0	
raise_r_o_eyebrow	SFInt32	0100011	0100011	0100011		[-I, +I]	0	
squeeze_l_eyebrow	SFInt32	0100100	0100100	0100100		[-I, +I]	0	
squeeze_r_eyebrow	SFInt32	0100101	0100101	0100101		[-I, +I]	0	
puff_l_cheek	SFInt32	0100110	0100110	0100110		[-I, +I]	0	
puff_r_cheek	SFInt32	0100111	0100111	0100111		[-I, +I]	0	
lift_l_cheek	SFInt32	0101000	0101000	0101000		[0, +I]	0	
lift_r_cheek	SFInt32	0101001	0101001	0101001		[0, +I]	0	
shift_tongue_tip	SFInt32	0101010	0101010	0101010		[-I, +I]	0	
raise_tongue_tip	SFInt32	0101011	0101011	0101011		[-I, +I]	0	

thrust_tongue_tip	SFInt32	0101100	0101100	0101100		[-I, +I]	0	
raise_tongue	SFInt32	0101101	0101101	0101101		[-I, +I]	0	
tongue_roll	SFInt32	0101110	0101110	0101110		[0, +I]	0	
head_pitch	SFInt32	0101111	0101111	0101111		[-I, +I]	0	
head_yaw	SFInt32	0110000	0110000	0110000		[-I, +I]	0	
head_roll	SFInt32	0110001	0110001	0110001		[-I, +I]	0	
lower_t_midlip_o	SFInt32	0110010	0110010	0110010		[-I, +I]	0	
raise_b_midlip_o	SFInt32	0110011	0110011	0110011		[-I, +I]	0	
stretch_l_cornerlip	SFInt32	0110100	0110100	0110100		[-I, +I]	0	
stretch_r_cornerlip	SFInt32	0110101	0110101	0110101		[-I, +I]	0	
lower_t_lip_lm_o	SFInt32	0110110	0110110	0110110		[-I, +I]	0	
lower_t_lip_rm_o	SFInt32	0110111	0110111	0110111		[-I, +I]	0	
raise_b_lip_lm_o	SFInt32	0111000	0111000	0111000		[-I, +I]	0	
raise_b_lip_rm_o	SFInt32	0111001	0111001	0111001		[-I, +I]	0	
raise_l_cornerlip_o	SFInt32	0111010	0111010	0111010		[-I, +I]	0	
raise_r_cornerlip_o	SFInt32	0111011	0111011	0111011		[-I, +I]	0	
stretch_l_nose	SFInt32	0111100	0111100	0111100		[-I, +I]	0	
stretch_r_nose	SFInt32	0111101	0111101	0111101		[-I, +I]	0	
raise_nose	SFInt32	0111110	0111110	0111110		[-I, +I]	0	
bend_nose	SFInt32	0111111	0111111	0111111		[-I, +I]	0	
raise_l_ear	SFInt32	1000000	1000000	1000000		[-I, +I]	0	
raise_r_ear	SFInt32	1000001	1000001	1000001		[-I, +I]	0	
pull_l_ear	SFInt32	1000010	1000010	1000010		[-I, +I]	0	
pull_r_ear	SFInt32	1000011	1000011	1000011		[-I, +I]	0	

H.1.34 FBA

FBA	SFWorldNode SF3DNode					0100010 001010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
face	SFFaceNode	0	0	0				
body	SFBodyNode	1	1	1				

H.1.35 FDP

FDP	SFWorldNode SFFDPNode					0100011 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
featurePointsCoord	SFCoordinateNode	00	00	00				
textureCoord	SFTextureCoordinateNode	01	01	01				
faceDefTables	MFFaceDefTablesNode	10	10	10				
faceSceneGraph	MF3DNode	11	11	11				

H.1.36 FIT

FIT	SFWorldNode SFFITNode					0100100 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
FAPs	MFInt32	0000	0000	0000		[-1, 68]	13 7	
Graph	MFInt32	0001	0001	0001		[0, 68]	13 7	
numeratorExp	MFInt32	0010	0010	0010		[0, 15]	13 4	
denominatorExp	MFInt32	0011	0011	0011		[0, 15]	13 4	
numeratorImpulse	MFInt32	0100	0100	0100		[0, 1023]	13 10	
numeratorTerms	MFInt32	0101	0101	0101		[0, 10]	13 4	
denominatorTerms	MFInt32	0110	0110	0110		[0, 10]	13 4	
numeratorCoefs	MFFloat	0111	0111	0111		[-I, +I]		
denominatorCoefs	MFFloat	1000	1000	1000		[-I, +I]		

H.1.37 Face

Face	SFWorldNode SFFaceNode					0100101 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
fit	SFFITNode	00	00	00				
fdp	SFFDPNode	01	01	01				
fap	SFFAPNode	10	10	10				
url	MFURL							
renderedFace	MF3DNode	11	11	11				

H.1.38 FaceDefMesh

FaceDefMesh	SFWorldNode SFFaceDefMeshNode					0100110 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
faceSceneGraphNode	SF3DNode	00						
intervalBorders	MFInt32	01					0	
coordIndex	MFInt32	10					0	
displacements	MFVec3f	11					0	

H.1.39 FaceDefTables

FaceDefTables	SFWorldNode SFFaceDefTablesNode					0100111 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
fapID	SFInt32	00				[1, 68]	13 7	
highLevelSelect	SFInt32	01				[1, 64]	13 6	
faceDefMesh	MFFaceDefMeshNode	10	0	0				
faceDefTransform	MFFaceDefTransformNode	11	1	1				

H.1.40 FaceDefTransform

FaceDefTransform	SFWorldNode SFFaceDefTransformNode					0101000 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
faceSceneGraphNode	SF3DNode	000						
fieldId	SFInt32	001						
rotationDef	SFRotation	010					10	
scaleDef	SFVec3f	011					1	
translationDef	SFVec3f	100					1	

H.1.41 FontStyle

FontStyle	SFWorldNode SFFontStyleNode					0101001 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
family	MFString	0000						
horizontal	SFBool	0001						
justify	MFString	0010						
language	SFString	0011						
leftToRight	SFBool	0100						
size	SFFloat	0101			0	[0, +I]	11	7

spacing	SFFloat	0110			1	[0, +I]	11	7
style	SFString	0111						
topToBottom	SFBool	1000						

H.1.42 Form

Form	SFWorldNode SF2DNode					0101010 00110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF2DNode		00					
removeChildren	MF2DNode		01					
children	MF2DNode	00	10	0				
size	SFVec2f	01	11	1		[0, +I]	12	1
groups	MFInt32	10				[0, 1023]	13 10	
constraint	MFInt32	11				[0, 255]	13 8	

H.1.43 Group

Group	SFWorldNode SFTopNode SF3DNode					0101011 01 001011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF3DNode		00					
removeChildren	MF3DNode		01					
children	MF3DNode	00	10					
bboxCenter	SFVec3f	01				[-I, +I]	1	
bboxSize	SFVec3f	10				[0, +I]	11	

H.1.44 Group2D

Group2D	SFWorldNode SFTopNode SF2DNode					0101100 10 00111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF2DNode		00					
removeChildren	MF2DNode		01					
children	MF2DNode	00	10					
bboxCenter	SFVec2f	01				[-I, +I]	2	
bboxSize	SFVec2f	10				[-I, +I]	12	

H.1.45 Image2D

Image2D	SFWorldNode SF2DNode					0101101 01000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***

url	MFURL							

H.1.46 ImageTexture

ImageTexture	SFWorldNode SFTextureNode					0101110 011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
url	MFURL	00						
repeatS	SFBool	01						
repeatT	SFBool	10						

H.1.47 IndexedFaceSet

IndexedFaceSet	SFWorldNode SFGeometryNode					0101111 01000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_colorIndex	MFInt32		000					
set_coordIndex	MFInt32		001					
set_normalIndex	MFInt32		010					
set_texCoordIndex	MFInt32		011					
color	SFColorNode	0000	100	00				
coord	SFCoordinateNode	0001	101	01				
normal	SFNormalNode	0010	110	10				
texCoord	SFTextureCoordinate Node	0011	111	11				
ccw	SFBool	0100						
colorIndex	MFInt32	0101				[-1, +I]	0	
colorPerVertex	SFBool	0110						
convex	SFBool	0111						
coordIndex	MFInt32	1000				[-1, +I]	0	
creaseAngle	SFFloat	1001				[0, 6.2831853]	6	
normalIndex	MFInt32	1010				[-1, +I]	0	
normalPerVertex	SFBool	1011						
solid	SFBool	1100						
texCoordIndex	MFInt32	1101				[-1, +I]	0	

H.1.48 IndexedFaceSet2D

IndexedFaceSet2D	SFWorldNode SFGeometryNode					0110000 01001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_colorIndex	MFInt32		000					
set_coordIndex	MFInt32		001					
set_texCoordIndex	MFInt32		010					

color	SFColorNode	000	011	00				
coord	SFCoordinate2DNode	001	100	01				
texCoord	SFTextureCoordinateNode	010	101	10				
colorIndex	MFInt32	011				[0, +I]	0	
colorPerVertex	SFBool	100						
convex	SFBool	101						
coordIndex	MFInt32	110				[0, +I]	0	
texCoordIndex	MFInt32	111				[0, +I]	0	

H.1.49 IndexedLineSet

IndexedLineSet	SFWorldNode SFGeometryNode					0110001 01010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_colorIndex	MFInt32		00					
set_coordIndex	MFInt32		01					
color	SFColorNode	000	10	0				
coord	SFCoordinateNode	001	11	1				
colorIndex	MFInt32	010				[-1, +I]	0	
colorPerVertex	SFBool	011						
coordIndex	MFInt32	100				[-1, +I]	0	

H.1.50 IndexedLineSet2D

IndexedLineSet2D	SFWorldNode SFGeometryNode					0110010 01011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_colorIndex	MFInt32		00					
set_coordIndex	MFInt32		01					
color	SFColorNode	000	10	0				
coord	SFCoordinate2DNode	001	11	1				
colorIndex	MFInt32	010				[0, +I]	0	
colorPerVertex	SFBool	011						
coordIndex	MFInt32	100				[0, +I]	0	

H.1.51 Inline

Inline	SFWorldNode SF3DNode SFStreamingNode					0110011 001100 100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
url	MFURL	00						
bboxCenter	SFVec3f	01				[-I, +I]	1	

bboxSize	SFVec3f	10				[0, +I]	11	

H.1.52 Inline2D

Inline2D	SFWorldNode SF2DNode SFStreamingNode					0110100 01001 101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
url	MFURL	00						
bboxCenter	SFVec2f	01				[-I, +I]	2	
bboxSize	SFVec2f	10				[-I, +I]	12	

H.1.53 LOD

LOD	SFWorldNode SF3DNode					0110101 001101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
level	MF3DNode	0000	0000	000				
center	SFVec3f	0001				[-I, +I]	1	
range	MFFloat	0010				[0, +I]	11	
addChildren	MF2DNode		0001					
removeChildren	MF2DNode		0010					
addChildrenLayer	MF2DNode		0011					
removeChildrenLayer	MF2DNode		0100					
children	MF2DNode	0011	0101	001				
childrenLayer	MFLayerNode	0100	0110	010				
size	SFVec2f	0101	0111	011	0	[-I, +I]	2	1
translation	SFVec2f	0110	1000	100	1	[-I, +I]	2	1
depth	SFInt32	0111	1001	101		[-I, +I]	3	
bboxCenter	SFVec2f	1000				[-I, +I]	2	
bboxSize	SFVec2f	1001				[-I, +I]	12	

H.1.54 Layer3D

Layer3D	SFWorldNode SFTopNode SFLayerNode					0110110 11 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF3DNode		0000					
removeChildren	MF3DNode		0001					

addChildren	MF3DNode		0010					
removeChildren	MF3DNode		0011					
children	MF3DNode	0000	0100	0000				
childrenLayer	MFLayerNode	0001	0101	0001				
translation	SFVec2f	0010	0110	0010	0	[-I, +I]	2	1
depth	SFInt32	0011	0111	0011		[-I, +I]	3	
size	SFVec2f	0100	1000	0100	1	[-I, +I]	2	1
background	SF3DNode	0101	1001	0101				
fog	SF3DNode	0110	1010	0110				
navigationInfo	SF3DNode	0111	1011	0111				
viewpoint	SF3DNode	1000	1100	1000				
bboxCenter	SFVec3f	1001				[-I, +I]	1	
bboxSize	SFVec3f	1010				[0, +I]	11	

H.1.55 Layout

Layout	SFWorldNode SF2DNode					0110111 01010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF2DNode		0000					
removeChildren	MF2DNode		0001					
children	MF2DNode	0000	0010	0000				
wrap	SFBool	0001	0011	0001				
size	SFVec2f	0010	0100	0010	00	[0, +I]	12	1
horizontal	SFBool	0011	0101	0011				
justify	MFString	0100	0110	0100				
leftToRight	SFBool	0101	0111	0101				
topToBottom	SFBool	0110	1000	0110				
spacing	SFFloat	0111	1001	0111	01	[0, +I]	0	7
smoothScroll	SFBool	1000	1010	1000				
loop	SFBool	1001	1011	1001				
scrollVertical	SFBool	1010	1100	1010				
scrollRate	SFFloat	1011	1101	1011	10	[-I, +I]	0	7

H.1.56 LineProperties

LineProperties	SFWorldNode SFLinePropertiesNode					0111000 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
lineColor	SFColor	00	00	00	0	[0, 1]	4	2
lineStyle	SFInt32	01	01	01		[0, 5]	13 3	
width	SFFloat	10	10	10	1	[0, +I]	12	7

H.1.57 ListeningPoint

ListeningPoint	SFWorldNode SF3DNode					0111001 001110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_bind	SFBool		00					
jump	SFBool	00	01	000				
orientation	SFRotation	01	10	001	0		10	6
position	SFVec3f	10	11	010	1	[-I, +I]	1	0
description	SFString	11						
bindTime	SFTime			011				
isBound	SFBool			100				

H.1.58 Material

Material	SFWorldNode SFMaterialNode					0111010 01		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
ambientIntensity	SFFloat	000	000	000	000	[0, 1]	4	7
diffuseColor	SFColor	001	001	001	001	[0, 1]	4	2
emissiveColor	SFColor	010	010	010	010	[0, 1]	4	2
shininess	SFFloat	011	011	011	011	[0, 1]	4	7
specularColor	SFColor	100	100	100	100	[0, 1]	4	2
transparency	SFFloat	101	101	101	101	[0, 1]	4	7

H.1.59 Material2D

Material2D	SFWorldNode SFMaterialNode					0111011 10		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
emissiveColor	SFColor	00	00	00	0	[0, 1]	4	2
filled	SFBool	01	01	01				
lineProps	SFLinePropertiesNode	10	10	10				
transparency	SFFloat	11	11	11	1	[0, 1]	4	7

H.1.60 MediaTimeSensor

MediaTimeSensor	SFWorldNode SF3DNode SF2DNode					0111100 001111 01011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
media	SFStreamingNode	0						
timer	SFTimeSensorNode	1						

H.1.61 MovieTexture

MovieTexture	SFWorldNode SFTextureNode SFStreamingNode					0111101 100 110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
loop	SFBool	000	000	000				
speed	SFFloat	001	001	001		[-I, +I]	0	7
startTime	SFTime	010	010	010		[-I, +I]		
stopTime	SFTime	011	011	011		[-I, +I]		
url	MFURL	100	100	100				
repeatS	SFBool	101						
repeatT	SFBool	110						
duration_changed	SFTime			101				
isActive	SFBool			110				

H.1.62 Normal

Normal	SFWorldNode SFNormalNode					0111110 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
vector	MFVec3f						9	4

H.1.63 NormalInterpolator

NormalInterpolator	SFWorldNode SF3DNode					0111111 010000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFVec3f	1	10	01		[-I, +I]	9	
value_changed	MFVec3f			10				

H.1.64 OrientationInterpolator

OrientationInterpolator	SFWorldNode SF3DNode					1000000 010001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFRotation	1	10	01		[-I, +I]	10	
value_changed	SFRotation			10				

H.1.65 PlaneSensor2D

PlaneSensor2D	SFWorldNode	1000001

	SF2DNode					01100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
autoOffset	SFBool	000	000	000				
enabled	SFBool	001	001	001				
maxPosition	SFVec2f	010	010	010		[-I, +I]	2	
minPosition	SFVec2f	011	011	011		[-I, +I]	2	
offset	SFVec2f	100	100	100		[-I, +I]	12	
trackPoint_changed	SFVec2f			101				

H.1.66 PointLight

PointLight	SFWorldNode SF3DNode					1000010 010010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
ambientIntensity	SFFloat	000	000	000	000	[0, 1]	4	7
attenuation	SFVec3f	001	001	001		[0, +I]	0	
color	SFColor	010	010	010	001	[0, 1]	4	2
intensity	SFFloat	011	011	011	010	[0, 1]	4	7
location	SFVec3f	100	100	100	011	[-I, +I]	1	0
on	SFBool	101	101	101				
radius	SFFloat	110	110	110	100	[0, +I]	11	7

H.1.67 PointSet

PointSet	SFWorldNode SFGeometryNode					1000011 01100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
color	SFColorNode	0	0	0				
coord	SFCoordinateNode	1	1	1				

H.1.68 PointSet2D

PointSet2D	SFWorldNode SFGeometryNode					1000100 01101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
color	SFColorNode	0	0	0				
coord	SFCoordinate2DNode	1	1	1				

H.1.69 Position2DInterpolator

Position2DInterpolator	SFWorldNode SF2DNode					1000101 01101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					

key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFVec2f	1	10	01		[-I, +I]	2	
value_changed	SFVec2f			10				

H.1.70 PositionInterpolator

PositionInterpolator	SFWorldNode SF3DNode					1000110 010011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFVec3f	1	10	01		[-I, +I]	1	
value_changed	SFVec3f			10				

H.1.71 Proximity2DSensor

Proximity2DSensor	SFWorldNode SF2DNode					1000111 01110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
center	SFVec2f	00	00	000		[-1, +I]	2	
size	SFVec2f	01	01	001		[0, +I]	12	
enabled	SFBool	10	10	010				
isActive	SFBool			011				
position_changed	SFVec2f			100				
orientation_changed	SFFloat			101				
enterTime	SFTime			110				
exitTime	SFTime			111				

H.1.72 ProximitySensor

ProximitySensor	SFWorldNode SF3DNode					1001000 010100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
center	SFVec3f	00	00	000		[-I, +I]	1	
size	SFVec3f	01	01	001		[0, +I]	11	
enabled	SFBool	10	10	010				
isActive	SFBool			011				
position_changed	SFVec3f			100				
orientation_changed	SFRotation			101				
enterTime	SFTime			110				
exitTime	SFTime			111				

H.1.73 QuantizationParameter

QuantizationParam	SFWorldNode					1001001		

eter	SF2DNode SF3DNode					01111 010101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
isLocal	SFBool	000000						
position3DQuant	SFBool	000001						
position3DMin	SFVec3f	000010				[-I, +I]	0	
position3DMax	SFVec3f	000011				[-I, +I]	0	
position3DNbBits	SFInt32	000100				[1, 32]	13 5	
position2DQuant	SFBool	000101						
position2DMin	SFVec2f	000110				[-I, +I]	0	
position2DMax	SFVec2f	000111				[-I, +I]	0	
position2DNbBits	SFInt32	001000				[1, 32]	13 5	
drawOrderQuant	SFBool	001001						
drawOrderMin	SFVec3f	001010				[-I, +I]	0	
drawOrderMax	SFVec3f	001011				[-I, +I]	0	
drawOrderNbBits	SFInt32	001100				[1, 32]	13 5	
colorQuant	SFBool	001101						
colorMin	SFFloat	001110				[0, 1]	0	
colorMax	SFFloat	001111				[0, 1]	0	
colorNbBits	SFInt32	010000				[1, 32]	13 5	
textureCoordinateQuant	SFBool	010001						
textureCoordinateMin	SFFloat	010010				[0, 1]	0	
textureCoordinateMax	SFFloat	010011				[0, 1]	0	
textureCoordinateNbBits	SFInt32	010100				[1, 32]	13 5	
angleQuant	SFBool	010101						
angleMin	SFFloat	010110				[0, 6.2831853]	0	
angleMax	SFFloat	010111				[0, 6.2831853]	0	
angleNbBits	SFInt32	011000				[1, 32]	13 5	
scaleQuant	SFBool	011001						
scaleMin	SFFloat	011010				[0, +I]	0	
scaleMax	SFFloat	011011				[0, +I]	0	
scaleNbBits	SFInt32	011100				[1, 32]	13 5	
keyQuant	SFBool	011101						
keyMin	SFFloat	011110				[-I, +I]	0	
keyMax	SFFloat	011111				[-I, +I]	0	
keyNbBits	SFInt32	100000				[-I, +I]	13 5	
normalQuant	SFBool	100001						
normalNbBits	SFInt32	100010				[1, 32]	13 5	

H.1.74 Rectangle

Rectangle	SFWorldNode SFGeometryNode					1001010 01110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
size	SFVec2f					[0, +I]	12	1

H.1.75 ScalarInterpolator

ScalarInterpolator	SFWorldNode SF3DNode SF2DNode					1001011 010110 10000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_fraction	SFFloat		00					
key	MFFloat	0	01	00		[0, 1]	8	
keyValue	MFFloat	1	10	01		[-I, +I]	0	
value_changed	SFFloat			10				

H.1.76 Shape

Shape	SFWorldNode SF3DNode SF2DNode					1001100 010111 10001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
appearance	SFAppearanceNode	0	0	0				
geometry	SFGeometryNode	1	1	1				

H.1.77 Sound

Sound	SFWorldNode SF3DNode					1001101 011000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
direction	SFVec3f	0000	0000	0000		[-I, +I]	9	
intensity	SFFloat	0001	0001	0001		[0, 1]	13 16	
location	SFVec3f	0010	0010	0010	000	[-I, +I]	1	0
maxBack	SFFloat	0011	0011	0011	001	[0, +I]	11	7
maxFront	SFFloat	0100	0100	0100	010	[0, +I]	11	7
minBack	SFFloat	0101	0101	0101	011	[0, +I]	11	7
minFront	SFFloat	0110	0110	0110	100	[0, +I]	11	7
priority	SFFloat	0111	0111	0111		[0, 1]	13 16	
source	SFAudioNode	1000	1000	1000				
spatialize	SFBool	1001						

H.1.78 Sound2D

Sound2D	SFWorldNode	1001110

	SF2DNode					10010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
intensity	SFFloat	00	00	00		[0, 1]	13 16	
location	SFVec2f	01	01	01		[-I, +I]	2	1
source	SFAudioNode	10	10	10				
spatialize	SFBool	11						

H.1.79 Sphere

Sphere	SFWorldNode SFGeometryNode					1001111 01111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
radius	SFFloat					[0, +I]	11	7

H.1.80 SpotLight

SpotLight	SFWorldNode SF3DNode					1010000 011001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
ambientIntensity	SFFloat	0000	0000	0000	0000	[0, 1]	4	7
attenuation	SFVec3f	0001	0001	0001	0001	[0, +I]	11	0
beamWidth	SFFloat	0010	0010	0010	0010	[0, 1.5707963]	6	3
color	SFColor	0011	0011	0011	0011	[0, 1]	4	2
cutOffAngle	SFFloat	0100	0100	0100	0100	[0, 1.5707963]	6	3
direction	SFVec3f	0101	0101	0101	0101	[-I, +I]	9	4
intensity	SFFloat	0110	0110	0110	0110	[0, 1]	4	7
location	SFVec3f	0111	0111	0111	0111	[-I, +I]	1	0
on	SFBool	1000	1000	1000				
radius	SFFloat	1001	1001	1001	1000	[0, +I]	11	7

H.1.81 Switch

Switch	SFWorldNode SF3DNode					1010001 011010		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
choice	MF3DNode	0	0	0				
whichChoice	SFInt32	1	1	1		[0, 1023]	13 10	

H.1.82 Switch2D

Switch2D	SFWorldNode SF2DNode					1010010 10011		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***

choice	MF2DNode	0	0	0				
whichChoice	SFInt32	1	1	1		[0, 1023]	13 10	

H.1.83 TermCap

TermCap	SFWorldNode SFGeometryNode					1010011 10000		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
evaluate	SFTime		0					
capability	SFInt		1	0		[0, 127]	13 8	
value	SFInt			1				

H.1.84 Text

Text	SFWorldNode SFGeometryNode					1010100 10001		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
string	SFString	00	00	00				
length	MFFloat	01	01	01		[0, +I]	11	
fontStyle	SFFontStyleNode	10	10	10				
maxExtent	SFFloat	11	11	11		[0, +I]	11	7

H.1.85 TextureCoordinate

TextureCoordinate	SFWorldNode SFTextureCoordinateNode					1010101 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
point	MFVec2f					[-I, +I]	5	1

H.1.86 TextureTransform

TextureTransform	SFWorldNode SFTextureTransformNode					1010110 1		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
center	SFVec2f	00	00	00	00	[-I, +I]	2	1
rotation	SFFloat	01	01	01	01	[0, 6.2831853]	6	7
scale	SFVec2f	10	10	10	10	[-I, +I]	7	1
translation	SFVec2f	11	11	11	11	[-I, +I]	2	1

H.1.87 TimeSensor

TimeSensor	SFWorldNode SFTimeSensorNode SF3DNode	1010111 1 011011

	SF2DNode					10100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
cycleInterval	SFTime	000	000	0000		[0, +I]		
enabled	SFBool	001	001	0001				
loop	SFBool	010	010	0010				
startTime	SFTime	011	011	0011		[-I, +I]		
stopTime	SFTime	100	100	0100		[-I, +I]		
cycleTime	SFTime			0101				
fraction_changed	SFFloat			0110				
isActive	SFBool			0111				
time	SFTime			1000				

H.1.88 TouchSensor

TouchSensor	SFWorldNode SF2DNode SF3DNode					1011000 10101 011100		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
enabled	SFBool			000				
hitNormal_changed	SFVec3f			001				
hitPoint_changed	SFVec3f			010				
hitTexCoord_changed	SFVec2f			011				
isActive	SFBool			100				
isOver	SFBool			101				
touchTime	SFTime			110				

H.1.89 Transform

Transform	SFWorldNode SF3DNode					1011001 011101		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF3DNode		000					
removeChildren	MF3DNode		001					
center	SFVec3f	000	010	000	000	[-I, +I]	1	0
children	MF3DNode	001	011	001				
rotation	SFRotation	010	100	010	001		10	6
scale	SFVec3f	011	101	011	010	[0, +I]	7	5
scaleOrientation	SFRotation	100	110	100	011		10	6
translation	SFVec3f	101	111	101	100	[-I, +I]	1	0
bboxCenter	SFVec3f	110				[-I, +I]	1	
bboxSize	SFVec3f	111				[0, +I]	11	

H.1.90 Transform2D

Transform2D	SFWorldNode SF2DNode					1011010 10110		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
addChildren	MF2DNode		0000					
removeChildren	MF2DNode		0001					
children	MF2DNode	0000	0010	000				
center	SFVec2f	0001	0011	001	000	[-I, +I]	2	1
rotationAngle	SFFloat	0010	0100	010	001	[0, 6.2831853]	6	3
scale	SFVec2f	0011	0101	011	010	[0, +I]	7	5
scaleOrientation	SFFloat	0100	0110	100	011	[0, 6.2831853]	6	3
drawingOrder	SFFloat	0101	0111	101		[0, +I]	3	
translation	SFVec2f	0110	1000	110	100	[-I, +I]	2	1
bboxCenter	SFVec2f	0111				[-I, +I]	2	
bboxSize	SFVec2f	1000				[-I, +I]	12	

H.1.91 Valuator

Valuator	SFWorldNode SF3DNode SF2DNode					1011011 011110 10111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
inSFBool	SFBool		00000					
inSFColor	SFColor		00001					
inMFColor	MFColor		00010					
inSFFloat	SFFloat		00011					
inMFFloat	MFFloat		00100					
inSFInt32	SFInt32		00101					
inMFInt32	MFInt32		00110					
inSFRotation	SFRotation		00111					
inMFRotation	MFRotation		01000					
inSFString	SFString		01001					
inMFString	MFString		01010					
inSFTime	SFTime		01011					
inSFVec2f	SFVec2f		01100					
inMFVec2f	MFVec2f		01101					
inSFVec3f	SFVec3f		01110					
inMFVec3f	MFVec3f		01111					
outSFBool	SFBool	0000	10000	0000				
outSFColor	SFColor	0001	10001	0001		[0, 1]	4	
outMFColor	MFColor	0010	10010	0010		[0, 1]	4	
outSFFloat	SFFloat	0011	10011	0011		[-I, +I]	0	
outMFFloat	MFFloat	0100	10100	0100		[-I, +I]	0	

outSFInt32	SFInt32	0101	10101	0101		[-I, +I]	0	
outMFInt32	MFInt32	0110	10110	0110		[-I, +I]	0	
outSFRotation	SFRotation	0111	10111	0111			10	
outMFRotation	MFRotation	1000	11000	1000			10	
outSFString	SFString	1001	11001	1001				
outMFString	MFString	1010	11010	1010				
outSFTime	SFTime	1011	11011	1011		[0, +I]		
outSFVec2f	SFVec2f	1100	11100	1100		[-I, +I]	2	
outMFVec2f	MFVec2f	1101	11101	1101		[-I, +I]	2	
outSFVec3f	SFVec3f	1110	11110	1110		[-I, +I]	1	
outMFVec3f	MFVec3f	1111	11111	1111		[-I, +I]	1	

H.1.92 VideoObject2D

VideoObject2D	SFWorldNode SF2DNode SFStreamingNode					1011100 11000 111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
loop	SFBool	000	000	000				
speed	SFFloat	001	001	001		[0, +I]	0	
startTime	SFTime	010	010	010				
stopTime	SFTime	011	011	011				
url	MFURL	100	100	100				
duration_changed	SFFloat			101				
isActive	SFBool			110				

H.1.93 Viewpoint

Viewpoint	SFWorldNode SF3DNode					1011101 011111		
Field name	***Field type***	***DEF id***	***IN id***	***OUT id***	***DYN id***	***[m, M]***	***Q***	***A***
set_bind	SFBool		000					
fieldOfView	SFFloat	000	001	000	00	[0, 3.1415927]	6	3
jump	SFBool	001	010	001				
orientation	SFRotation	010	011	010	01		10	6
position	SFVec3f	011	100	011	10	[-I, +I]	1	0
description	SFString	100						
bindTime	SFTime			100				
isBound	SFBool			101				

H.1.94 Viseme

Viseme	SFWorldNode SFVisemeNode					1011110 1		

Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
viseme_select1	SFInt32	00	00	00		[0, 31]	13 5	
viseme_select2	SFInt32	01	01	01		[0, 31]	13 5	
viseme_blend	SFInt32	10	10	10		[0, 63]	13 6	
viseme_def	SFBool	11	11	11				

H.1.95 WorldInfo

WorldInfo	SFWorldNode SF2DNode SF3DNode					1011111 11001 100000		
Field name	*Field type*	*DEF id*	*IN id*	*OUT id*	*DYN id*	*[m, M]*	*Q*	*A*
info	MFString	0						
title	SFString	1						

H.2 Node Data Type Tables

The Node Data Type (NDT) tables contain the following parameters:

- The name of NDT.
- The number of nodes in the NDT.
- The number of bits used to identify a node type within this NDT.
- The ID for each node type contained in this NDT.
- Number of DEF, IN, OUT and DYN fields as well as the number of bits used to encode them.

The table template is as follows:

Node Definition Type Tables

Node Data Type		**Number of nodes**			
Node type	ID	DEF	IN	OUT	DYN

H.2.1 SF2DNode

SF2DNode		**25 Nodes**			
reserved	00000				
AnimationStream	00001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Background2D	00010	0 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
ColorInterpolator	00011	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Conditional	00100	0 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
DiscSensor	00101	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits
Form	00110	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Group2D	00111	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Image2D	01000	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Inline2D	01001	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

Layout	01010	4 DEF bits	4 IN bits	4 OUT bits	2 DYN bits
MediaTimeSensor	01011	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
PlaneSensor2D	01100	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Position2DInterpolator	01101	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Proximity2DSensor	01110	2 DEF bits	2 IN bits	3 OUT bits	0 DYN bits
QuantizationParameter	01111	6 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ScalarInterpolator	10000	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Shape	10001	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Sound2D	10010	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Switch2D	10011	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
TimeSensor	10100	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits
TouchSensor	10101	0 DEF bits	0 IN bits	3 OUT bits	0 DYN bits
Transform2D	10110	4 DEF bits	4 IN bits	3 OUT bits	3 DYN bits
Valuator	10111	4 DEF bits	5 IN bits	4 OUT bits	0 DYN bits
VideoObject2D	11000	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
WorldInfo	11001	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.2 SF3DNode

SF3DNode		**32 Nodes**			
reserved	000000				
AnimationStream	000001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Background	000010	4 DEF bits	4 IN bits	4 OUT bits	2 DYN bits
Billboard	000011	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Collision	000100	3 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
ColorInterpolator	000101	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
CompositeMap	000110	1 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Conditional	000111	0 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
CoordinateInterpolator	001000	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
DirectionalLight	001001	3 DEF bits	3 IN bits	3 OUT bits	2 DYN bits
FBA	001010	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Group	001011	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Inline	001100	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
LOD	001101	4 DEF bits	4 IN bits	3 OUT bits	1 DYN bits
ListeningPoint	001110	2 DEF bits	2 IN bits	3 OUT bits	1 DYN bits
MediaTimeSensor	001111	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
NormalInterpolator	010000	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
OrientationInterpolator	010001	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
PointLight	010010	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
PositionInterpolator	010011	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
ProximitySensor	010100	2 DEF bits	2 IN bits	3 OUT bits	0 DYN bits
QuantizationParameter	010101	6 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ScalarInterpolator	010110	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Shape	010111	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Sound	011000	4 DEF bits	4 IN bits	4 OUT bits	3 DYN bits

SpotLight	011001	4 DEF bits	4 IN bits	4 OUT bits	4 DYN bits
Switch	011010	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
TimeSensor	011011	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits
TouchSensor	011100	0 DEF bits	0 IN bits	3 OUT bits	0 DYN bits
Transform	011101	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
Valuator	011110	4 DEF bits	5 IN bits	4 OUT bits	0 DYN bits
Viewpoint	011111	3 DEF bits	3 IN bits	3 OUT bits	2 DYN bits
WorldInfo	100000	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.3 SFAppearanceNode

SFAppearanceNode		**1 Node**			
reserved	0				
Appearance	1	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits

H.2.4 SFAudioNode

SFAudioNode		**6 Nodes**			
reserved	000				
AudioClip	001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
AudioDelay	010	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
AudioFX	011	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
AudioMix	100	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
AudioSource	101	3 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
AudioSwitch	110	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits

H.2.5 SFBodyNode

SFBodyNode		**1 Node**			
reserved	0				
Body	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.6 SFColorNode

SFColorNode		**1 Node**			
reserved	0				
Color	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.7 SFCoordinate2DNode

SFCoordinate2DNode		**1 Node**			
reserved	0				
Coordinate2D	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.8 SFCoordinateNode

SFCoordinateNode		1 Node			
reserved	0				
Coordinate	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.9 SFExpressionNode

SFExpressionNode		1 Node			
reserved	0				
Expression	1	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits

H.2.10 SFFAPNode

SFFAPNode		1 Node			
reserved	0				
FAP	1	7 DEF bits	7 IN bits	7 OUT bits	0 DYN bits

H.2.11 SFFDPNode

SFFDPNode		1 Node			
reserved	0				
FDP	1	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits

H.2.12 SFFITNode

SFFITNode		1 Node			
reserved	0				
FIT	1	4 DEF bits	4 IN bits	4 OUT bits	0 DYN bits

H.2.13 SFFaceDefMeshNode

SFFaceDefMeshNode		1 Node			
reserved	0				
FaceDefMesh	1	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.14 SFFaceDefTablesNode

SFFaceDefTablesNode		1 Node			
reserved	0				
FaceDefTables	1	2 DEF bits	1 IN bits	1 OUT bits	0 DYN bits

H.2.15 SFFaceDefTransformNode

SFFaceDefTransformNode		1 Node			
reserved	0				
FaceDefTransform	1	3 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.16 SFFaceNode

SFFaceNode		1 Node			
reserved	0				
Face	1	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits

H.2.17 SFFontStyleNode

SFFontStyleNode		1 Node			
reserved	0				
FontStyle	1	4 DEF bits	0 IN bits	0 OUT bits	1 DYN bits

H.2.18 SFGeometryNode

SFGeometryNode		17 Nodes			
reserved	00000				
Box	00001	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Circle	00010	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Cone	00011	2 DEF bits	0 IN bits	0 OUT bits	1 DYN bits
Curve2D	00100	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Cylinder	00101	3 DEF bits	0 IN bits	0 OUT bits	1 DYN bits
ElevationGrid	00110	4 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Extrusion	00111	4 DEF bits	2 IN bits	0 OUT bits	2 DYN bits
IndexedFaceSet	01000	4 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedFaceSet2D	01001	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedLineSet	01010	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
IndexedLineSet2D	01011	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
PointSet	01100	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
PointSet2D	01101	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Rectangle	01110	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Sphere	01111	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
TermCap	10000	0 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Text	10001	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits

H.2.19 SFLayerNode

SFLayerNode		1 Node			
reserved	0				
Layer3D	1	4 DEF bits	4 IN bits	4 OUT bits	1 DYN bits

H.2.20 SFLinePropertiesNode

SFLinePropertiesNode		1 Node			
reserved	0				
LineProperties	1	2 DEF bits	2 IN bits	2 OUT bits	1 DYN bits

H.2.21 SFMaterialNode

SFMaterialNode		2 Nodes			
reserved	00				
Material	01	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
Material2D	10	2 DEF bits	2 IN bits	2 OUT bits	1 DYN bits

H.2.22 SFNormalNode

SFNormalNode		1 Node			
reserved	0				
Normal	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.23 SFStreamingNode

SFStreamingNode		7 Nodes			
reserved	000				
AnimationStream	001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
AudioClip	010	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
AudioSource	011	3 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Inline	100	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Inline2D	101	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
MovieTexture	110	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
VideoObject2D	111	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits

H.2.24 SFTextureCoordinateNode

SFTextureCoordinateNode		1 Node			
reserved	0				
TextureCoordinate	1	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits

H.2.25 SFTextureNode

SFTextureNode		4 Nodes			
reserved	000				
Composite2DTexture	001	2 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
Composite3DTexture	010	3 DEF bits	4 IN bits	3 OUT bits	0 DYN bits

ImageTexture	011	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
MovieTexture	100	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits

H.2.26 SFTextureTransformNode

SFTextureTransformNode		**1 Node**			
reserved	0				
TextureTransform	1	2 DEF bits	2 IN bits	2 OUT bits	2 DYN bits

H.2.27 SFTimeSensorNode

SFTimeSensorNode		**1 Node**			
reserved	0				
TimeSensor	1	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits

H.2.28 SFTopNode

SFTopNode		**3 Nodes**			
reserved	00				
Group	01	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Group2D	10	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Layer3D	11	4 DEF bits	4 IN bits	4 OUT bits	1 DYN bits

H.2.29 SFVisemeNode

SFVisemeNode		**1 Node**			
reserved	0				
Viseme	1	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits

H.2.30 SFWorldNode

SFWorldNode		**95 Nodes**			
reserved	0000000				
AnimationStream	0000001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Appearance	0000010	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
AudioClip	0000011	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
AudioDelay	0000100	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
AudioFX	0000101	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
AudioMix	0000110	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
AudioSource	0000111	3 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
AudioSwitch	0001000	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Background	0001001	4 DEF bits	4 IN bits	4 OUT bits	2 DYN bits
Background2D	0001010	0 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Billboard	0001011	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits

Body	0001100	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Box	0001101	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Circle	0001110	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Collision	0001111	3 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Color	0010000	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ColorInterpolator	0010001	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Composite2DTexture	0010010	2 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
Composite3DTexture	0010011	3 DEF bits	4 IN bits	3 OUT bits	0 DYN bits
CompositeMap	0010100	1 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Conditional	0010101	0 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Cone	0010110	2 DEF bits	0 IN bits	0 OUT bits	1 DYN bits
Coordinate	0010111	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Coordinate2D	0011000	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
CoordinateInterpolator	0011001	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Curve2D	0011010	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Cylinder	0011011	3 DEF bits	0 IN bits	0 OUT bits	1 DYN bits
DirectionalLight	0011100	3 DEF bits	3 IN bits	3 OUT bits	2 DYN bits
DiscSensor	0011101	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits
ElevationGrid	0011110	4 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Expression	0011111	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Extrusion	0100000	4 DEF bits	2 IN bits	0 OUT bits	2 DYN bits
FAP	0100001	7 DEF bits	7 IN bits	7 OUT bits	0 DYN bits
FBA	0100010	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
FDP	0100011	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
FIT	0100100	4 DEF bits	4 IN bits	4 OUT bits	0 DYN bits
Face	0100101	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
FaceDefMesh	0100110	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
FaceDefTables	0100111	2 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
FaceDefTransform	0101000	3 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
FontStyle	0101001	4 DEF bits	0 IN bits	0 OUT bits	1 DYN bits
Form	0101010	2 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Group	0101011	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Group2D	0101100	2 DEF bits	2 IN bits	0 OUT bits	0 DYN bits
Image2D	0101101	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ImageTexture	0101110	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
IndexedFaceSet	0101111	4 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedFaceSet2D	0110000	3 DEF bits	3 IN bits	2 OUT bits	0 DYN bits
IndexedLineSet	0110001	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
IndexedLineSet2D	0110010	3 DEF bits	2 IN bits	1 OUT bits	0 DYN bits
Inline	0110011	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Inline2D	0110100	2 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
LOD	0110101	4 DEF bits	4 IN bits	3 OUT bits	1 DYN bits
Layer3D	0110110	4 DEF bits	4 IN bits	4 OUT bits	1 DYN bits
Layout	0110111	4 DEF bits	4 IN bits	4 OUT bits	2 DYN bits
LineProperties	0111000	2 DEF bits	2 IN bits	2 OUT bits	1 DYN bits

ListeningPoint	0111001	2 DEF bits	2 IN bits	3 OUT bits	1 DYN bits
Material	0111010	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
Material2D	0111011	2 DEF bits	2 IN bits	2 OUT bits	1 DYN bits
MediaTimeSensor	0111100	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
MovieTexture	0111101	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Normal	0111110	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
NormalInterpolator	0111111	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
OrientationInterpolator	1000000	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
PlaneSensor2D	1000001	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
PointLight	1000010	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
PointSet	1000011	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
PointSet2D	1000100	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Position2DInterpolator	1000101	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
PositionInterpolator	1000110	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Proximity2DSensor	1000111	2 DEF bits	2 IN bits	3 OUT bits	0 DYN bits
ProximitySensor	1001000	2 DEF bits	2 IN bits	3 OUT bits	0 DYN bits
QuantizationParameter	1001001	6 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
Rectangle	1001010	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
ScalarInterpolator	1001011	1 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Shape	1001100	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Sound	1001101	4 DEF bits	4 IN bits	4 OUT bits	3 DYN bits
Sound2D	1001110	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
Sphere	1001111	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
SpotLight	1010000	4 DEF bits	4 IN bits	4 OUT bits	4 DYN bits
Switch	1010001	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Switch2D	1010010	1 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
TermCap	1010011	0 DEF bits	1 IN bits	1 OUT bits	0 DYN bits
Text	1010100	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
TextureCoordinate	1010101	0 DEF bits	0 IN bits	0 OUT bits	0 DYN bits
TextureTransform	1010110	2 DEF bits	2 IN bits	2 OUT bits	2 DYN bits
TimeSensor	1010111	3 DEF bits	3 IN bits	4 OUT bits	0 DYN bits
TouchSensor	1011000	0 DEF bits	0 IN bits	3 OUT bits	0 DYN bits
Transform	1011001	3 DEF bits	3 IN bits	3 OUT bits	3 DYN bits
Transform2D	1011010	4 DEF bits	4 IN bits	3 OUT bits	3 DYN bits
Valuator	1011011	4 DEF bits	5 IN bits	4 OUT bits	0 DYN bits
VideoObject2D	1011100	3 DEF bits	3 IN bits	3 OUT bits	0 DYN bits
Viewpoint	1011101	3 DEF bits	3 IN bits	3 OUT bits	2 DYN bits
Viseme	1011110	2 DEF bits	2 IN bits	2 OUT bits	0 DYN bits
WorldInfo	1011111	1 DEF bits	0 IN bits	0 OUT bits	0 DYN bits