ETSI TS 103 223 V1.1.1 (2015-04)

TECHNICAL SPECIFICATION

MDA;
Object-Based Audio Immersive Sound Metadata and Bitstream

EBU
OPERATING EUROVISION

Reference
DTS/JTC-027

Keywords
audio, broadcast, contribution

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://ipr.etsi.org).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

NOTE: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document specifies the object model, reference renderer, bitstream syntax and broadcast extensions for MDA. MDA, short for Multi-Dimension Audio, is a metadata model and bitstream representation of an object-based soundfield for linear content, for use in cinema and broadcast applications.

The presentdocument consists of four main clauses. The metadata clause (Clause 4) provides a metadata model independent of (bitstream) representation, with a strong emphasis on cinematic content. Clause 5 specifies a reference renderer, providing semantics for the MDA metadata model. Clause 6 specifies a preferred bitstream representation of the MDA metadata model. Note that the metadata model allows for more than one bitstream representation. Finally, Clause 7 specifies an extension of the core MDA model to include metadata and bitstream elements specifically suited for broadcast content. This Clause includes among others metadata for Loudness, Higher Order Ambisonics and Interactivity.

Unless otherwise stated, MDA metadata are specified using Unified Modeling Language [4].

Note that the MDA core metadata, reference renderer and bitstream documents have been submitted to SMPTE 25CSS "Immersive Sound Model and Bitstream" [i.2] for consideration towards an interoperable immersive sound model and bitstream for cinematographic linear content.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] IETF RFC 3986 (January 2005): "Uniform Resource Identifier (URI): Generic Syntax".

[2] Recommendation ITU-R BS.1770-3: "Algorithms to measure audio programme loudness and true-peak audio level".

[3] Recommendation ITU-R BS.1771-1: "Requirements for loudness and true-peak indicating meters", January 2012.

[4] ISO/IEC 19501 (2005): "Information technology -- Open Distributed Processing -- Unified Modeling Language (UML) Version 1.4.2".

[5] EBU Tech - 3342: "Loudness Range: A measure to supplement loudness normalization in accordance with EBU R 128".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the reference document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] Pulkki, Ville, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning", JAES Volume 45 Issue 6 pp. 456-466; June 1997.

[i.2] MDA Bitstream Specification, SMPTE 25CSS Interoperable Immersive Sound, January 2014.

[i.3] The Ambisonics Association.

NOTE See: http://ambisonics.ch/.

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

aperture: circular extent of an audio object, measured in degrees

azimuth: angle in degrees between the frontal direction and the position of an audio object projected on the horizontal plane

divergence: horizontal spread of an audio object, measured in degrees

elevation: angle in degrees between the horizontal plane and the position of an MDA audio object (viewed as a vector from origin to the object position)

frame: independently decodable fragment of an MDA bitstream

loudness: measure for the perceived energy (loudness) of an audio stream during playback

namespace: identified set of identifiers

renderer: algorithm/method for producing sound with an MDA audio bitstream as input

slice: segment of an MDA bitstream corresponding to an interval in time for which MDA metadata are constant

soundfield: identified set of audio reproduction devices, located at standardized positions

3.2 Abbreviations

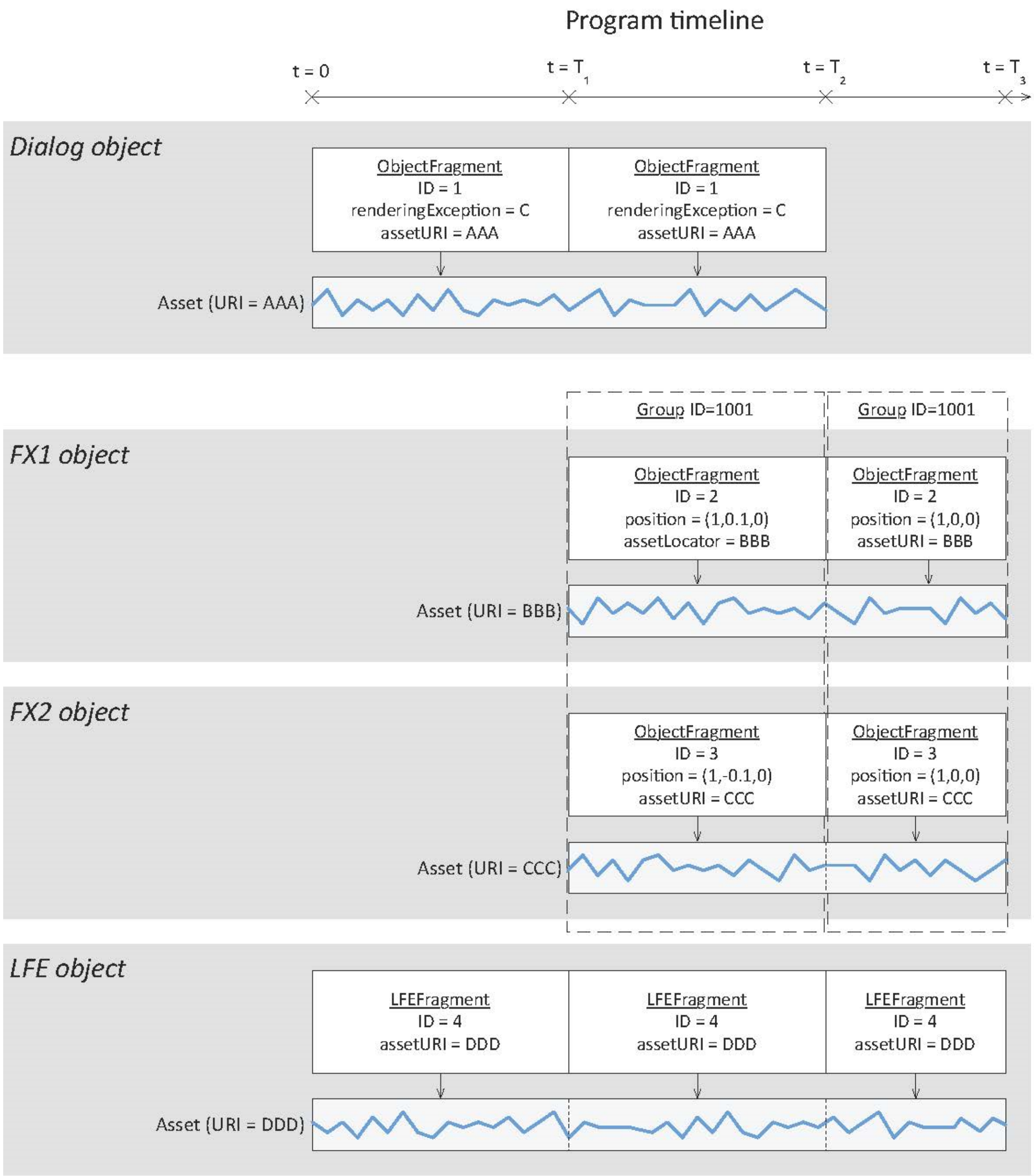
For the purposes of the present document, the following abbreviations apply:

ACN	Ambisonics Channel Number
ATSC	Advanced Television Systems Committee
BPX	Broadcast Extension
CRC	Cyclic Redundancy Check
DRC	Dynamic Range Compression
DRM	Digital Rights Management
HOA	Higher Order Ambisonics
LFE	Low Frequency Effects
LKFS	Loudness, K-Weighted, relative to Full Scale
LSB	Least Significant Bit
LU	Loudness Unit
LUFS	Loudness Units relative to Full Scale
MDA	Multi-Dimensional Audio
MSB	Most Significant Bit
PCM	Pulse-code Modulation
RTIL	Real-Time Instantaneous Loudness
SMPTE	Society of Motion Picture and Television Engineers
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF8	Universal Character Set + Transformation Format-**8**-bit
VBAP	Vector Based Amplitude Panning

4 MDA Core Metadata

4.1 Introduction (informative)

The MDA Program, or simply Program hereafter, is a self-contained object-based audio program. As such it consists of a collection of audio objects, each combining an audio waveform with metadata. The metadata indicates, for instance, when the object occurs on the Program timeline or where it is positioned within the soundfield. It is used to control the mapping of the audio object waveform to output loudspeakers at playback.



NOTE: Only a subset of all Fragment properties are shown.

Figure 4.1: Sample Program

Figure 4.1 from t = 0 to t = T_3, the dialog object exists only from t = 0 to t = T_2, and the FX2 and FX2 objects from t = T_1 to t = T_3. The Program object model allows for any number of audio objects to overlap at any point in time, and an object can be as short as a sample or as long as the program.

Objects are identified by object identification numbers, which are unique for the duration of an MDA Program. The metadata associated with each object is divided into Fragments, each corresponding to a period of time during which the metadata is static, referred to as a Slice. Typically multiple Objects live within the same Slice To simplify the Program structure, boundaries are aligned.

Two kinds of Fragments, and hence audio objects, are defined:

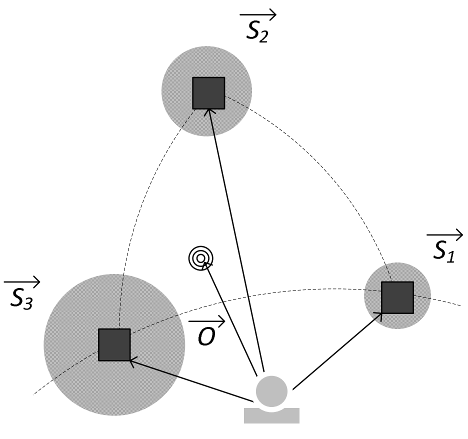
- An ObjectFragment corresponds to an object associated with a spatial locus. For instance, the positions of the FX1 and FX2 objects in Figure 4.1 changes from $t = T_1$ to $t = T_3$. This spatial locus is used to determine the loudspeakers that will output the waveform associated with the object. It is also possible to instruct that an object waveform be routed through a specific loudspeaker, if present.

- An LFEFragment corresponds to an object whose waveform is intended for routing to a Low Frequency Effect (LFE) channel, and is therefore not associated with a spatial location.

Each Fragment references a sequence of audio samples, i.e. the object waveform, within an underlying asset identified using a Uniform Resource Identifier (URI) [1]. Depending on applications, the asset can be carried alongside the Fragment metadata or be remote. Multiple Fragments can reference the same audio samples within a single asset.

- As illustrated by the FX1 and FX2 objects, Fragments can be combined into a Group, which logically groups the Fragments and contains metadata common to the Fragments. The object model also allows Fragments to be combined into a Switch, which indicates that only one of the Fragments is rendered at any given time. Groups and Switches are recursive entities that can themselves contain Groups and Switches. From the perspective of the object model, Fragments, Groups and Switches are all subclasses of the Entity class, which represents arbitrary entities of the Program timeline. It is not required by the MDA core specification that clustering of fragments into Groups and Switches is consistent over the lifetime of objects (as in Figure 4.1), but applications can require this as an additional constraint.

- In order to specify unambiguously how object metadata is used to map object waveform to loudspeaker outputs, i.e. rendered, a Reference Renderer is fully specified in Clause 5. The Reference Renderer uses the Vector Base Amplitude Panning formalism (VBAP), which was introduced by Pulkki et al. [i.1] and has since been extensively studied. VBAP is an extension of the familiar tangent law for pair-wise panning to three-dimensional speaker configurations. Specifically, given a loudspeaker triplet on the unit sphere and a point source object located within the spherical triangle defined by the loudspeakers, the contribution of the object waveform to each of the loudspeaker is determined by the coordinates of the object within the linear basis formed by the three loudspeakers (see Figure 4.2). Objects with a finite extent can be rendered as a collection of point sources. More complex loudspeaker configurations can be decomposed into multiple speaker triplets.



NOTE: The shaded areas show the relative output power at each of the speakers and are determined by expressing the object vector in the basis formed by the three loudspeakers.

Figure 4.2: Rendering audio objects using VBAP

To support a range of applications within its stated scope, the Program object model is designed to be flexible, e.g. the number of simultaneous Fragments is not limited, and offers multiple extension points. Applications are therefore expected to constrain or extend the object model to suit their specific requirements. Similarly, the present document does not define a concrete representation of the Program, and mappings to bitstream structures and transmission mechanisms are left to other documents.

4.2 Timeline

An MDA Program defines a sample-accurate timeline onto which Entity instances are placed (see Figure 4.7).

Positions and durations on the timeline shall be expressed as integer multiples of the inverse of the Program audio sample rate (see Clause 4.5.5.3), i.e. as an integer number of audio samples. In other words, the granularity of the timeline is the audio sample. The origin of the timeline (t=0) is arbitrary.

4.3 Audio Objects

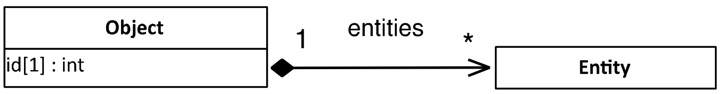


Figure 4.3: MDA Audio Object

An MDA Audio Object represents an identifiable sequence of Entity instances (see Clause 4.5.6) within the time span of an MDA Program. It is uniquely identified by its (identification) number `id`, and aggregates all of the MDA Entity instances in the MDA Program with that same identification number. Two Entity instances are said to belong to the same Object if and only if they have the same `id` value.

4.4 Coordinate System

The present document uses the Cartesian coordinate system illustrated in Figure 4.4 and specified as follows:

- the listener is located at the origin O=(0,0,0), facing the front of the room;
- the positive z-axis is perpendicular to the floor of the room, and directed to the ceiling;
- the positive y-axis is directed towards the front of the room;
- the positive x-axis is directed to the right of the listener;
- loudspeakers lie on the unit sphere S; and
- the unit circle in the x-y plane is the locus of traditional horizontal two-dimensional loudspeaker configurations.

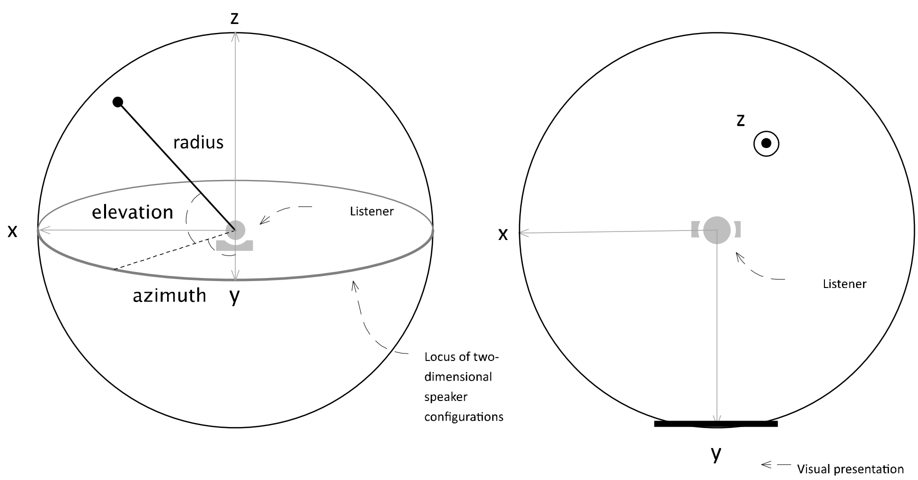


Figure 4.4: Program Coordinate System

For convenience, the following modified spherical coordinate system is also defined:

$$x = \rho \sin\theta \cos\varphi$$
$$y = \rho \cos\theta \cos\varphi$$
$$z = \rho \sin\varphi$$

where the symbols ρ (*rho*), θ (*theta*) and φ (*phi*) denote the radius, azimuth and elevation of the object, respectively.

4.5 Object Model

4.5.1 General

The Program object model is specified using a combination of prose and UML notation. The prose shall take precedence over the UML notation in case of conflict.

If an optional property is absent, its value shall be unspecified unless a default value is provided, in which case its value shall be the default value.

Values that are identified as reserved shall not be used in the present document and, if present in a Program, shall be ignored by implementations conforming to the present document.

The notation <SymbolName> refers to the URI constant with symbol SymbolName.

4.5.2 Namespace

UML elements defined herein shall be members of the MDA Package within the 1.0 core namespace specified in Table 4.1 and abbreviated as <mdacore>.

Table 4.1: MDA Object Model Namespaces

Symbol	URI
mdaroot	http://mdaif.org
mdacore	http://mdaif.org/core/1.0

4.5.3 Versioning

The namespace specified in Clause 4.5.2 **SHALL** only be associated with Program instances that conform to the present document as expressed by the combination of its prose and schema definitions. Program instances using specifications that modify the latter (i.e. schema), including future versions of the present document, **SHALL** use a different namespace.

4.5.4 Program

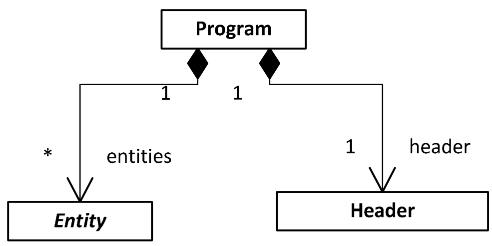


Figure 4.5: Program Object Model

A Program instance is a single complete Program, which contains all entities necessary for reproduction.

4.5.5 Header

4.5.5.1 General

The single Header instance **SHALL** contain information applicable to the Program as a whole.

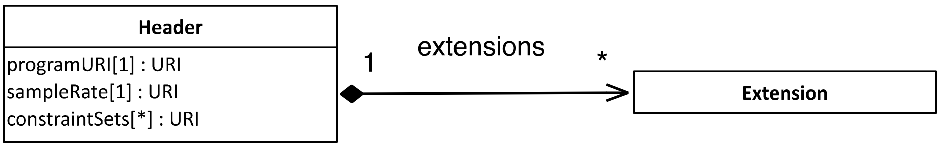


Figure 4.6: Header Model

4.5.5.2 programURI

The `programURI` property **SHALL** uniquely identify the Program instance. It shall consist of no more than 64 characters, with the meaning of character specified in RFC 3986 [1].

Two Program instances may have identical `programURI` values if and only if the two instances are identical.

4.5.5.3 sampleRate

The `sampleRate` property **SHALL** indicate the audio sampling rate of the Program.

Table 4.2 defines common values for audio sampling rates.

4.5.5.4 constraintSets

The Program object model may be constrained and extended by multiple applications, each potentially defining additional metadata properties and applying a set of constraints beyond those specified herein. Implementations can use the `constraintSets` to rapidly determine whether they are capable of processing a Program.

Each item of the `constraintSets` property **SHALL** be unambiguously associated with a collection of normative provisions (beyond those specified herein) to which the Program conforms.

No two items of the `constraintSets` property **SHALL** be equal.

4.5.5.5 extensions

The `extensions` property allows application-specific metadata (contained in a concrete subclass of the Extension class) to be associated with the Program.

4.5.6 Entities

4.5.6.1 General

Each Entity instance **SHALL** correspond to an entity on the Program timeline, associated with a start offset (inclusive) and an end offset (exclusive), defining the Entity Time Window. The end offset **SHALL** be larger than the start offset. The duration of an entity **SHALL** be the positive difference by end offset and start offset (see Figure 4.7).

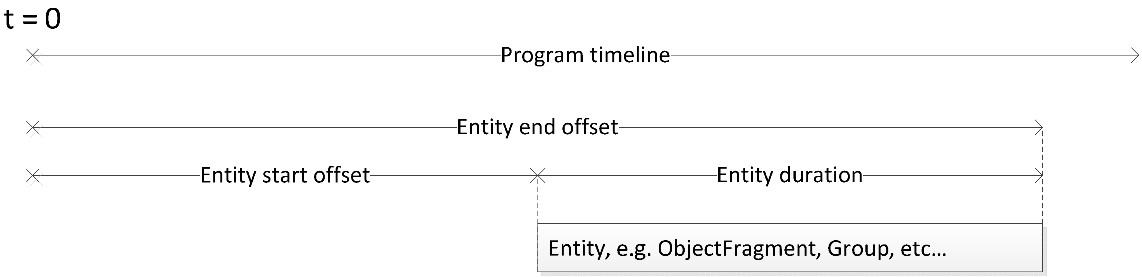


Figure 4.7: Positioning an Entity instance on the Program Timeline

The present document defines a number of concrete subclasses of the Entity class, and future revisions may define additional ones.

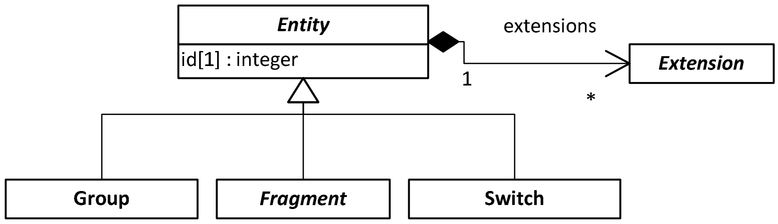


Figure 4.8: Entity Model

4.5.6.2 id

The `id` property allows multiple related Entities to be uniquely linked within the scope of the Program as belonging to the same Object (see Clause 4.3).

The following constraints apply:

- No two Entity instances belonging to the same Audio Object **SHALL** overlap on the timeline. In other words, an Audio Object **SHALL** be present at most once at any given moment in time.
- Each Entity instance belonging to the same Object **SHALL** have the same Type, referred to as the Type of the Object, where the Type of an Object is refers to the class of the Entity instance. For the prsent document, allowed Type names are `ObjectType`, `LFEType`, `GroupType` and `SwitchType`.
- The value of the `id` property **SHALL** belong to the range $[0, 2^{32}-1]$.

4.5.6.3 extensions

The `extensions` property allows application-specific metadata (contained in concrete subclasses of the Extension class defined by the application) to be associated with an Entity.

4.5.7 Group

4.5.7.1 General

A Group instance is a logical group of Entity instances. The Type of a Group instance is `GroupType`.

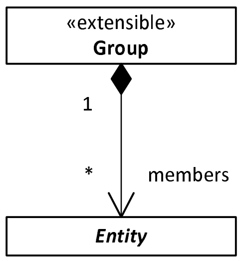


Figure 4.9: Group Model

The start offset of a Group instance **SHALL** be the smallest start offset of all Entity instances it contains.

The end offset of a Group instance **SHALL** be the largest end offset of all Entity instances it contains.

Note that all Entity instances within a Group instance shall be rendered at any given time (see Clause 5.3.1).

4.5.8 Switch

4.5.8.1 General

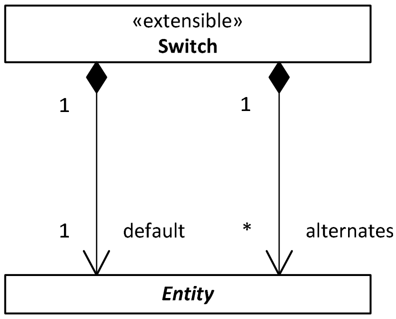


Figure 4.10: Switch Model

A Switch instance is a logical group of Entity instances, one of which **SHALL** be designated as `default` and the others designated as `alternates`. The Type of a Switch instance is `SwitchType`.

The start offset of a Switch instance **SHALL** be the smallest start offset of all Entity instances it contains.

The end offset of a Switch instance **SHALL** be the largest end offset of all Entity instances it contains.

Note that precisely one Entity instance within a Switch instance shall be rendered at any given time. This single instance shall be referenced by the `default` property unless otherwise specified by the rendering context (see Clause 5.3.1).

4.5.9 Fragment

4.5.9.1 General

A Fragment represents an Entity that spans a specified interval of the Program timeline.

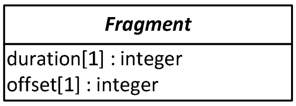


Figure 4.11: Fragment Object Model

The start offset of a Fragment instance shall be the value of its `offset` property.

The end offset of a Fragment instance shall be the sum of the values of its `offset` and `duration` properties.

4.5.9.2 offset property

The value of the `offset` property **SHALL** be in the range $[0, 2^{64}-1]$.

4.5.9.3 duration property

The duration `property` **SHALL** be in the range $[0, 2^{16}-1]$.

4.5.10 MonoSourceFragment

4.5.10.1 General

A `MonoSourceFragment` represents a monaural sound source whose characteristics are constant over the duration of the `MonoSourceFragment`.

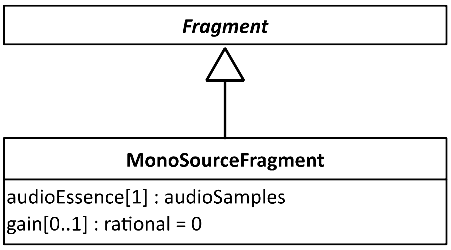


Figure 4.12: MonoSourceFragment Object Model

4.5.10.2 audioEssence

The audioEssence property **SHALL** reference the audio samples of the monaural sound source.

4.5.10.3 gain

The gain property specifies a gain that is applied to the source audio samples. It allows the relative gain of sources to be adjusted without modifying the latter. In particular, this property allows the re-use of audioEssence at different gain levels. Note that values larger than 0 may lead to clipping and need to be carefully applied.

The following constraints apply:

- The value of gain **SHALL** be in the range [-411,100]/4 in units of dB. The value -411/4 **SHALL** be interpreted as -INF, (negative infinity).

4.5.11 ObjectFragment

4.5.11.1 General

An ObjectFragment instance represents a monaural sound source that can be positioned within the soundfield. The Type of an ObjectFragment is ObjectType.

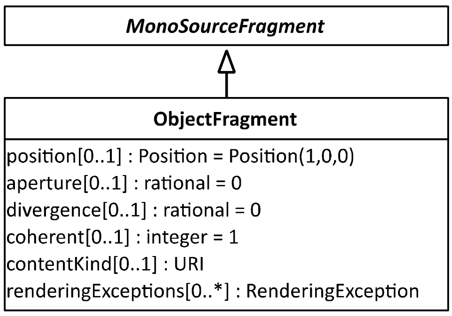


Figure 4.13: ObjectFragment Object Model

The extent of the ObjectFragment is parameterized by aperture and divergence values.

4.5.11.2 position

The position property **SHALL** be the position of the sound source.

4.5.11.3 aperture

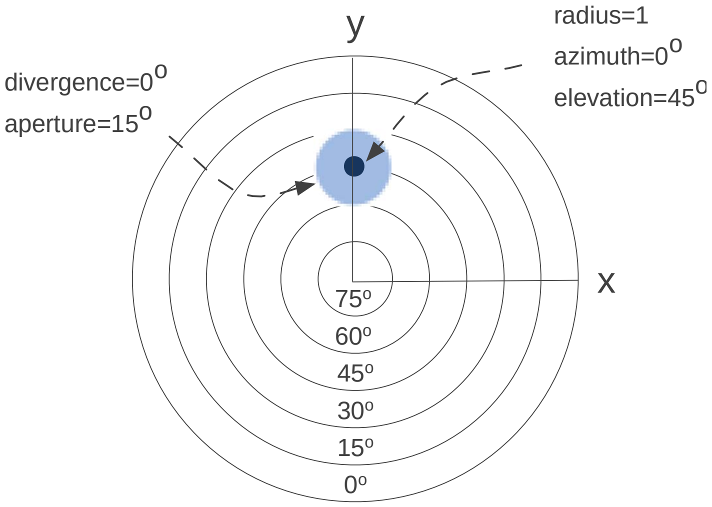


Figure 4.14: 15° aperture shown using stereographic projection

The `aperture` property shall be the nominal extent of the sound source before the application of divergence (see Clause 4.5.11.4). The nominal extent is the spherical cap defined by the intersection of (i) the sphere whose radius is the segment between the position of the sound source and the origin and (ii) the infinite right circular solid cone whose apex is at the origin and its aperture equal to twice the `aperture` property value. Figure 4.14 illustrates a 15° aperture using stereographic projection.

The following constraints apply:

- The `aperture` property **SHALL** be in the set [0,255]×(180/255) and **IS** in units of degrees. An aperture value of 180 degrees indicates that the source extent covers the entire sphere.

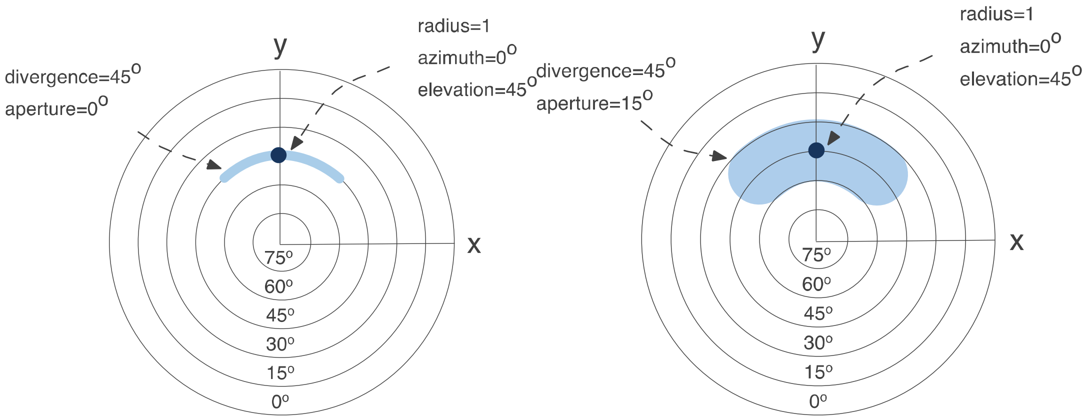


Figure 4.15: Combining Aperture and Divergence (stereographic projection)

4.5.11.4 divergence

The `divergence` property shall be the half angle of the horizontal arc (latitude) centered at the sound source location over which the nominal extent is spread to yield the source extent. Figure 4.15 illustrates a 45° divergence applied to a 0° and 15° aperture.

The following constraints apply:

- The `divergence` property **SHALL** be in the set [0,255]×(180/255) and is in units of degrees.

4.5.11.5 coherent

The `coherent` property **SHALL** indicate whether the source is rendered coherently (`coherent` equal to 1) or diffusively (`coherent` equal to 0) across its extent.

The following constraints apply:

- The value of `coherent` **SHALL** be in the set {0, 1}.

4.5.11.6 renderingExceptions

A `soundfieldName` **MAY** be associated with an MDA reference renderer instance and is typically set at renderer initialization (e.g. using a configuration file). If the `targetConfiguration` property of a `RenderingException` instance matches the target renderer `soundfieldName`, the normative MDA VBAP renderer is overridden, and the rendering method indicated in the `RenderingException` instance is executed.

`RenderingExceptions` are defined in Clause 4.5.14 and their semantics in the MDA reference renderer is specified in Clause 5.3.2.

The following constraints apply:

- Any `renderingException.targetConfiguration` value, including the NULL value, **SHALL** occur at most once in the `renderingException` instances in `renderingExceptions`. This constraint allows the MDA reference renderer to make an unambiguous choice of `RenderingException` for a given MDA entity (see Clause 5.3.2).

4.5.11.7 contentKind

The `contentKind` property indicates the audio content of the ObjectFragment. The present document defines three values (`<AudioContentKindDialog>, <AudioContentKindEffects>, and <AudioContentKindMusic>)`, but applications MAY add other values (see Table 4.2).

Implementations may ignore values they do not recognize.

4.5.12 LFEFragment

4.5.12.1 General

An LFEFragment instance represents a low-frequency effects sound source. The Type of an LFEFragment is `LFEType`. A `Slice` may contain 0, 1 or more `LFEFragment` instances. However, applications may restrict the number of allowed simultaneous `LFEFragment` instances.

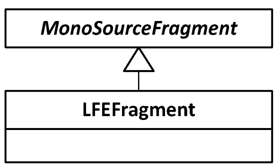


Figure 4.16: LFEFragment Object Model

4.5.13 AudioSamples

An AudioSamples instance selects a sequence of audio samples from an underlying asset.

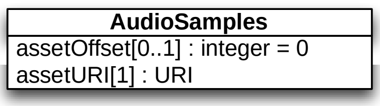


Figure 4.17: AudioSamples Object Model

4.5.13.1 assetOffset

The `assetOffset` property **SHALL** indicate the offset of the first audio sample selected within the asset referenced by the `assetURI` property.

The following constraints apply:

- The value of the `assetOffset` property **SHALL** be in the range [0, 2^{64}-1] in units of audio samples.

4.5.13.2 assetURI

The `assetURI` property shall reference a sequence of audio samples. The nature of the reference is left to specifications mapping the MDA Program to bitstreams. For instance, the reference can be either internal or external to the bit stream.

The following constraints apply:

- No two sequences of audio samples **SHALL** have identical `assetURI` values unless they are identical.

4.5.14 RenderingException

4.5.14.1 General

The `RenderingException` class allows the author to override default VBAP rendering behaviour in the presence of the rendering configurations listed in the `targetConfiguration` property.

Each concrete subclasses of the `RenderingException` class shall specify a specific rendering behaviour.

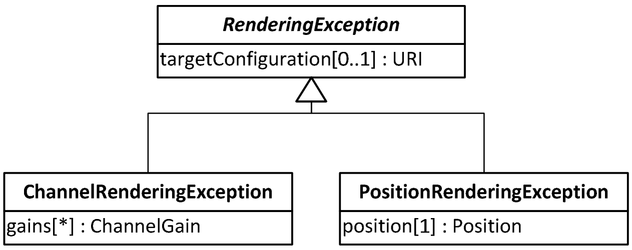


Figure 4.18: RenderingException Object Model

4.5.14.2 targetConfiguration property

The `targetConfiguration` property holds a valid `soundfieldName` or NULL (see Clause 5.2.1.1) and **SHALL** indicate the rendering configuration to which the `RenderingException` instance shall apply (see Clause 5.3.2)

4.5.15 PositionRenderingException

4.5.15.1 General

The `PositionRenderingException` allows the author to indicate an alternate position for the object, as specified by the `PositionRenderingException.position` property.

4.5.15.2 position

The `position` property **SHALL** be the alternate position of the `ObjectFragment` instance for which the `RenderingException` instance applies.

4.5.16 ChannelRenderingException

4.5.16.1 General

The `ChannelRenderingException` allows the author to explicitly specify the channels to which the object waveform is routed, when rendering to specified target rendering configuration.

4.5.16.2 gains

Each item of the `gains` property specifies an output channel (specified by `ChannelGain.channel`) to which the audio samples of the object are routed after applying a gain (specified by the `ChannelGain.gain`).

If the `gains` property contains no elements, the rendering of the `ObjectFragment` instance is suppressed.

The following constraints apply:

- Two different members of the `gains` property **SHALL NOT** have the same value for `ChannelGain.channel`.

4.5.17 ChannelGain

4.5.17.1 General

A `ChannelGain` instance associates a gain with an audio channel.

ChannelGain
gain[1] : rational channel[1] : URI

Figure 4.19: ChannelGain Object Model

4.5.17.2 gain property

The value of the `gain` property **SHALL** belong to the set [-255,0]/4 and **SHALL** be in units of dB.

4.5.17.3 channel property

The `channel` property indicates the audio channel to which the gain property applies. The LFE channel **SHOULD NOT** appear in channel rendering exceptions.

4.5.18 Extension

The `Extension` class is abstract. Application-specific metadata is added to the Program by defining concrete subclasses.

4.5.19 Position

4.5.19.1 General

A Position instance represents a position in the coordinate system specified in Clause 4.4.

Position
radius[0..1] : rational = 1 azimuth[0..1] : rational = 0 elevation[0..1] : rational = 0

Figure 4.20: Position Object Model

4.5.19.2 radius

The value of the `radius` property shall correspond to the ρ coordinate of the position, i.e. the radial distance.

The following constraints apply:

- The `radius` property **SHALL** take values in the set [0,4 095]/2 047.

4.5.19.3 azimuth

The `azimuth` property shall correspond to the θ coordinate of the position.

The following constraints apply:

- The value of the `azimuth` property **SHALL** be in the set [-2 048,2 047]×(180/2 048) in units of degrees (and in the corresponding set in units of radians).

4.5.19.4 elevation

The value of the `elevation` property **SHALL** correspond to the φ coordinate of the position.

The following constraints apply:

- The value of the `elevation` property **SHALL** be in the set [-1 023,1 023]×(90/1 023) in units of degrees (and in the corresponding set in units of radians).

4.6 Overall Constraints

4.6.1 Aligned Fragment Instances

Entity intervals **SHALL** be mutually disjoint, i.e. two different intervals **SHALL NOT** overlap.

4.7 URI Constants

Table 4.2 **SHALL** define URI constants used by the present document.

Table 4.2: URI Constants

Symbol	URI	Definition
Content Kind		
`AudioContentKindDialog`	`<mdacore>/labels/content-kind/dialog`	Any spoken words or narration, including unidentifiable human vocalizations e.g. crowd walla or cheers.
`AudioContentKindEffects`	`<mdacore>/labels/content-kind/effects`	Sound effects of any kind, including ambience and Foley.
`AudioContentKindMusic`	`<mdacore>/labels/content-kind/music`	Includes all underlying music elements, e.g. score and songs, as well as diegetic music from sources integral to the story, e.g. a concert or a radio.
Sampling Rate		
`AudioSampleRate48000`	`<mdacore>/labels/sample-rate/48000Hz`	Audio sampled at 48 000 Hz.
`AudioSampleRate96000`	`<mdacore>/labels/sample-rate/96000Hz`	Audio sampled at 96 000 Hz.

4.8 Basic Data Types

4.8.1 General

The present document makes use of the following data types. No assumption is made on their encoding.

4.8.2 Real

The real data type shall consist of all real numbers.

4.8.3 Rational

The rational data type shall consist of all rational numbers.

4.8.4 Integer

The integer data type shall consist of all integer numbers.

4.8.5 URI

The URI data type shall be a URI as specified in RFC 3986 [1].

5 MDA Reference Renderer

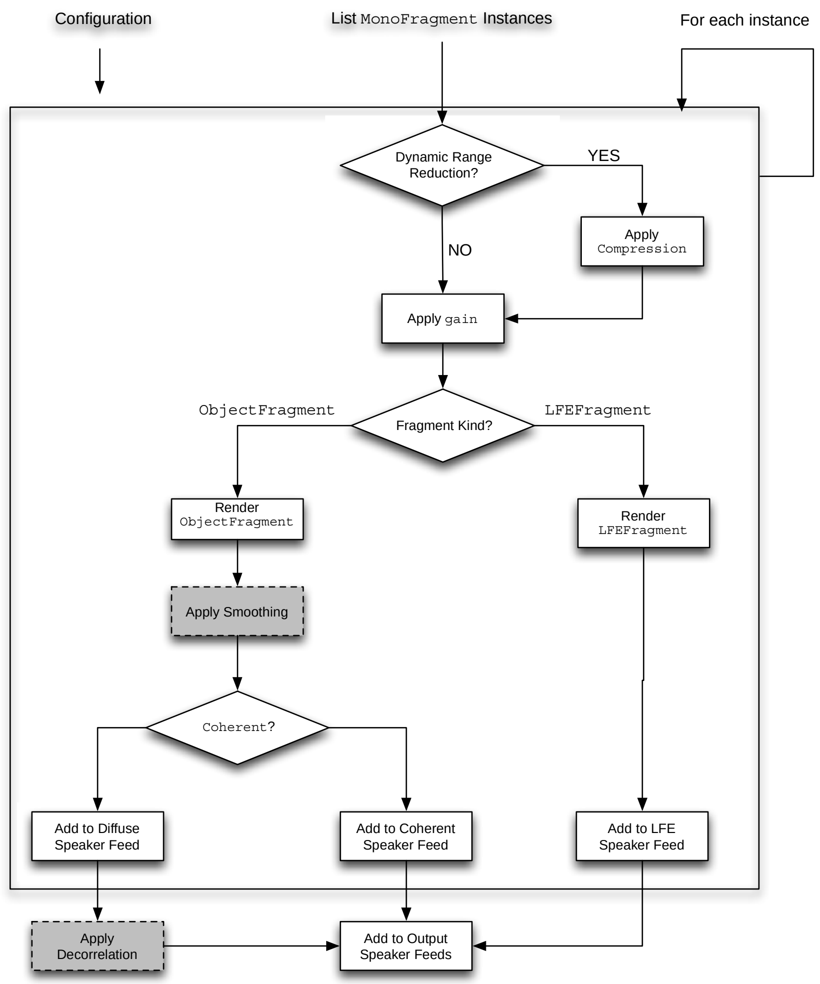
5.1 Overview

The reference renderer, illustrated in Figure 5.1, operates on successive offsets in the Program timeline. With the exception of two functions whose definition is outside of the scope of the present document and are free parameters of the MDA Reference Renderer, i.e. `ApplyDecorrelation` and `ApplySmoothing`, the reference renderer is stateless. It is encouraged that implementers of an MDA Reference Renderer select decorrelation and smoothing functions that meet expected quality and performance requirements.

A smoothing function that works quite well in practice is simple linear interpolation of rendering gains (see below). A simple but effective decorrelation method is achieved by using frequency dependedent random (small) delays per speaker feed, for example by means of a low order all-pass filter (see below). However, depending on context, other smoothing functions or decorrelators may be preferred.

```
/*
 * Function: simpleSmooth
 *
 * Input:
 *   g0 : gain value for previous slice
 *   g1 : gain value for current slice
 *   n  : sample index relative to start of current slice
 * Output
 *   simpleSmooth(n,g0,g1): smoothed gain value
 */

simpleSmooth(n,g0,g1) = { return ((duration-n)*g0 + n*g1) / duration; }

/*
 * Function: simpleDecor
 * Simple order 2 real all-pass filter.
 *
 * Input:
 *  in : in[s,n] non-decorrelated input signal for speaker s at time n.
 *
 * Output :
 *  out : out[s,n] decorrelated output for speaker s at time n.
 */

foreach s in speakers {

    pole[s] = RandomPole();

    a[s] = 2*Re(pole[s]);

    b[s] = |pole[s]|^2;

}

out[s,n] = simpleDecor(in,s,n) {

    return in[s,n-2] + a[s]*(out[s,n-1) - in[s,n-1]) + b[s]*(in[s,n] - out[s,n-2];

}
```

The reference renderer uses the rendering configuration information contained in configuration structure (see Clause 5.2) to render to speaker signals the MonosourceFragment instances that exists at each offset.



NOTE: The dotted line and grey-filled processes are not specified and left to applications.

Figure 5.1: Reference rendering process overview

5.2 Configuration

5.2.1 General

The `Configuration` structure holds the information necessary for the reference renderer to render `MonoSourceFragment` instances to loudspeakers. With the exception of `virtualSources`, which is generated, all information needs to be provided as input to the rendering process.

Following UML conventions, the absence of a property is indicated by the value NULL.

```
struct Configuration {

 URI            soundfieldName;
 NormalSpeaker  speakers[];
 LFESpeaker     lfe;
 Patch          patches[];
 VirtualSource  virtualSources[];

};
```

5.2.2 soundfieldName

If not NULL, Configuration.soundfieldName **SHALL** identify the speaker configuration embodied by the combination of Configuration.speakers and Configuration.lfe (see Clause 5.2.1.2).

As specified in Clause 4.5.11.6 and Clause 5.3.2, the Configuration.soundfieldName is used to match against the targetConfigurations property of a RenderingExceptions instance, i.e. used to determine whether or not a RenderingException instance applies.

5.2.3 Speakers

Two kinds of loudspeakers can be specified, normal and Low Frequency Effects (LFE), corresponding to Configuration.speakers and Configuration.lfe, respectively. An LFESpeaker is required to render LFEFragment instances. However, an LFESpeaker may be defined virtual (see below) with its input being rerouted to other Speaker instances (using mix coefficients, see below). Multiple LFE speakers may exists, and be distinguished by a standard interpretation of their identifying URI (e.g. urn:itu:bs:2051:0:speaker:LFE1 for a left side LFE speaker and urn:itu:bs:2051:0:speaker:LFE2 for a right side LFE speaker).

```
struct Speaker {

    URI name;
    float               mixcoefs[];

};

Struct NormalSpeaker : Speaker {

    Position            position;

}

Struct LFESpeaker : Speaker {
}
```

The combination of Configuration.speakers and Configuration.lfe speakers is the set of loudspeakers to which MonoSourceFragment instances are rendered.

If present, the Speaker.name property **SHALL** identify the speaker within the speaker configuration indicated by the soundfieldName property. This name is used when executing ChannelRenderingExceptions (see Clause 5.3.2).

The NormalSpeaker.position shall be the position of the speaker expressed in the coordinate system of Clause 4.5.19. In particular, NormalSpeaker.position.radius **SHALL** be equal to 1.

The NormalSpeaker.mixcoefs property is an array of gain values, each associated with the Speaker instance in Configuration.speakers with the same index. The value of the property speakers[m].mixcoefs[m] shall be either 0 or 1. In case of the former, the speaker with index "m" is referred to as a *virtual* speaker. In case of the latter, all other mix coefficients **SHALL** be 0, and the speaker is referred to as a *physical* speaker. The signal for a virtual speaker "m" is redistributed to other normal speakers "n" with gain equal to speakers[m].mixcoefs[n]. A normal speaker without a mixcoefs property **SHALL** be physical.

The following constraints apply:

- No two speakers **SHALL** have identical name values (unique identifier).
- No two speakers **SHALL** have identical position values (unique position).
- Virtual speakers **SHALL** only be distributed to physical speakers.

5.2.4 patches

```
typedef int   Patch [3];
```

A `Patch` instance is a triplet of speaker indices `T=(m[1],m[2],m[3])`, where each `m[i]` corresponds to a normal speaker position in the set of all speaker positions `H`. The `patches` property is an array of `Patch` instances.

Let `Plane[T]` be the plane defined by the speaker locations in a `Patch` instance `T` and let `HalfSpace[T]` be the unique closed half-space defined by `P[T]` that includes O. Then `T` is contained in `patches` if and only if all of the speaker positions `H` lie in `HalfSpace[T]` and `P[T]` does not contain the origin `O` (i.e. the speaker positions are linearly independent).

Given a set of normal speaker `H` positions, the naïve computation of `patches` is straightforward, albeit potentially slow.

5.2.5 Virtual Sources

```
struct VirtualSource {

    Position            position;
    float                   gains[];

};
```

To render extended sources, the reference renderer first renders a collection of point sources uniformly sampled over the unit sphere, and then averages over those virtual sources located within the source extent. The `virtualSources` property shall be computed according to the pseudo-code program below.

```
ComputeVirtualSources(Configuration config) {

    // Resolution

    int kElevationDivs = 64;            // -90 => +90
    int kAzimuthDivs = kElevationDivs * 2;  // 0 => 360

    // For all elevations

    for(int i = -kElevationDivs/2; i <= kElevationDivs/2; i++) {

        float elevation = i * 180/kElevationDivs;

        // Compute elevation adapted azimuth resolution

        int n = round(kAzimuthDivs * cosd(elevation));

        // Sources at the poles

        if (abs(i) == kElevationDivs/2) n = 1;

        // For all azimuths

        for(int j = 0; j < n ; j++) {

            float azimuth = 360 * j / n;

            // Create virtual source at position

            VirtualSource vs;

            vs.position.radius = 1;
            vs.position.azimuth = azimuth;
            vs.position.elevation = elevation;

            // Render virtual source

            int cnt = RenderPointSource(config, vs.position, vs.gains);

            // Add virtual source to list
            // if virtual source is rendered

            if (cnt > 0) {
                virtualSources.append(vs);
            }
```

```
        }
 }
}
```

The kElevationsDivs and kAzimuthDivs constants determine the number of virtual sources present on a given meridian and on the equator of the unit sphere, respectively.

5.3 Rendering Process

5.3.1 ProcessOffset

For every successive offset T within Program timeline, the reference renderer shall execute ProcessOffset(config, fragments, speakerOutput, lfeOutput), with:

- config as specified in Clause 5.2.
- fragments consisting of the MonoSourceFragment instances for which T lies within the time interval of MonoSourceFragment.
- Each element of speakerOutput corresponding to the signal to be output by the element in config.speakers with the same index.
- lfeOutput corresponding to the signal to be output by the LFE loudspeaker, if any.

In addition:

- For a given offset T, the reference rendered **SHALL** render all members of a Group Entity simultaneously.
- For a given offset T, the reference renderer **SHALL** render precisely one of the members of a Switch Entity. If not explicitly indicated which of the members is to be rendered, the default member of the Switch Entity **SHALL** be rendered.

```
ProcessOffset(  Configuration config,
                                        MonoSourceFragment fragments[],
                                        float &speakerOutput[],
                                        float &lfeOutput ) {

    // Speaker Buffers

    speakerOutput.fill(0);
    lfeOutput = 0;

    // Coherent and diffuse buffers

    float coherentOutput[speakerOutput.size()];
    float diffuseOutput[speakerOutput.size()];

    foreach(fragment in fragments) {

        float sample = GetCurrentAudioSample(fragment.audioSamples);

        // Apply gain

        sample *= pow(10, fragment.compression / 20);

        // Render

        if (fragment intstanceOf ObjectFragment) {

            if (fragment.coherent) {
                coherentOutput += sample * RenderObjectFragment(config, fragment);
            } else {
                diffuseOutput += sample * RenderObjectFragment(config, fragment);
            }
        } else if (fragment intstanceOf LFEFragment){
            if (config.lfe != NULL) {
                LFEOutput += sample;
            }
        }
    }
```

```
    // Apply diffusion

    ApplyDecorrelation(config, diffuseOutput);

    // Combine coherent and diffuse signals

    speakerOutput = coherentOutput + diffuseOutput;

}
```

`GetCurrentAudioSample(AudioSamples audioSample)` **SHALL** return the value of the audio sample of `audioSample` associated at the current timeline position.

`ApplySmoothing(config, id, gains)`, which is not specified here and is a free parameter to an MDA Reference Renderer, should minimize temporal transients for elements of gains, across `ObjectFragment` instances with the same `id`, and may preserve state across calls. `ApplySmoothing` should preserve normalization of the `gains` vector.

`ApplyDecorrelation(config, diffuseOutput)`, which is not specified here and is a free parameter to an MDA Reference Renderer, should minimize correlation across elements of `diffuseOutput` and may preserve state across calls.

5.3.2 Render Object Fragment

The `RenderObjectFragment` function renders an object with gains *normalized to unit power*. It returns the number of point sources used to rendering the object.

```
int RendeObjectFragment(
                        Configuration config,
                        ObjectFragment fragment,
                    float gains[]) {

    //assert: gains.size()=config.speakers.size()

    float output[config.speakers.size()];

    // Try to match to soundfieldName
    // Return NULL is no match found

    RenderingException selectedEx =
        fragment.renderingExceptions.match(config.soundfieldName);

    // Try to match (unique) RE with target is NULL
    // Succeeds if matching speaker set is found
    // Return NULL if no match found

    if (selectedEx == NULL) {
            selectedEx =
                fragment.renderingExceptions.match(config.speakers);
    };

    // Handle Channel Exception

    if (selectedEx instanceOf ChannelRenderingException) {

        foreach (gain in selectedEx.gains) {
            int i = config.speakers.index(s | s.name == gain.channel);
            gains[i] = gain.coef;

        }

    // Handle Position Exception
    // New position should satisfy applicable constraints.

    } else {

        if (selectedEx instanceOf PositionRenderingException) {
            fragment.position = selectedEx.position;
        }

        int rCount = RenderExtent(config, fragment, gains);

        // If number of virtual sources is small
        // fall back to point source
```

```
        if (rCount < 2) {
            rCount = RenderPointSource(config, fragment.position, gains);
        }
    }

    // Redistribute virtual speakers
    // Independent of how gains
    // were originally computed.

    for (int i = 0; i < gains.size(); i++) {
        if (config.speakers[i] instanceOf VirtualSpeaker) {
                for (int j = 0; j < gains.size(); j++) {
                        gains[j] += gains[i] * config.speakers[i].mixcoefs[j];
            }
        }
    }

  // Normalize gains to unit power

  return gains / gains.norm();

}
```

The polymorphic match function `fragment.renderingExceptions.match` is defined as:

1) A `RenderingException` instance with a non-NULL `targetConfiguration` property value matches an MDA renderer instance if and only if the `config.soundfieldName` property value is the same.

2) A `RenderingException` instance with an NULL `targetConfiguration property` value matches a renderer MDA reference renderer instance if and only if all speakers listed in the `RenderException` instance are present in `config.speakers`. In particular, a `PositionRenderingException` always matches.

3) When multiple `RenderingException` instances match, the `RenderingException` instance with a non-NULL `targetConfiguration` takes precedence (there can only be one, see Clause 4.5.11.6).

5.3.3 RenderPatch

`RenderPatch` returns the contribution of a source located at `cartesianPosition` to each element of a triplet `speakerPatch` of speakers according to the Vector Base Amplitude Panning (VBAP) formalism [i.1].

A patch `P` such that `RenderPatch(P,Q)` returns a non-zero gain vector is said to render position `Q`.

By construction, `RenderPatch` *normalizes the linear combination* of speaker positions to the unit sphere.

```
float[] RenderPatch(
                NormalSpeaker[3]    speakerPatch,
                float[3]            cartesianPosition
                ) {

 float[3] coefs = {0,0,0};

    Matrix m = Matrix(
                    speakerPatch[0].position.toCartesian(),
                    speakerPatch[1].position.toCartesian(),
                    speakerPatch[2].position.toCartesian()
    );

    if (m.det() == 0) return {0,0,0};

    gains = m.invert() * cartesianPosition;

 if (coefs[0] >= 0 && cos[1]>=0 && gains[2]>=0) {
            return gains;
    } else {
        return {0,0,0};
    }
}
```

5.3.4 Point Source Rendering

Applying RenderPatch for all available patches and subsequent averaging renders a point source. Only points in the convex hull of config.patches can be rendered. The number of patches used in rendering the point source is returned.

By construction, RenderPointSource *normalizes the linear combination* of speaker positions to the unit sphere.

```
int RenderPointSource(
            Configuration config,
            Position pos,
            float gains[]) {

    //assert: gains.size()=config.speakers.size()

    float   coefs[3];

    int count = 0;

    foreach(patch in config.patches) {

        // Render with selected patch

        coefs = RenderPatch(
            {
                config.speakers[patch[0]],
                config.speakers[patch[1]],
                config.speakers[patch[2]]
            },
            pos.toCartesian()
        );

        // If success, add to gains

        if (isPositive(coefs)) {
            gains[patch[0]] += coefs[0];
            gains[patch[1]] += coefs[1];
            gains[patch[2]] += coefs[2];

            // Remember number of patches

            count++;
        };
    }

    // Normalize for number of successful patches.

    if (count > 0) {
        foreach (gain in gains) {
            gain /= count;
        }
    }

    // Return rendering success

    return count;
}
```

5.4 Extent Rendering

Simultaneous rendering the pre-calculated virtual sources virtualSources of the configuration config renders an extended source. The function returns the number of virtual sources used for rendering.

Only the intersection of the extent of fragment and the convex hull of config.patches can and will be rendered.

```
int RenderExtent(
                        Configuration config,
                        ObjectFragment fragment
                        float gains[]) {

    // assert: gains.size() = config.speakers.size()

    VirtualSource vss[] =
```

```
        SelectSources (config.virtualSources, fragment);

    // Virtual source contribution

    foreach(vs in vss) {
        gains += vs.gains;
    }

    // Return the number of rendering
    // virtual sources in the extent

    return vss.size();
}
```

The function `SelectSources(virtualSources, fragment)` selects all elements of `virtualSources` that are within the extent of `fragment`, as defined in Clause 0.

6 MDA Core Bitstream

6.1 Introduction

The Bitstream consists of an ordered sequence of Packets. As illustrated in Figure 6.1, each Packet consists of a payload preceded by a Label field identifying the nature of the payload and a length field indicating the size of the payload. Packet are byte-aligned, but information within their payloads not necessarily so. Implementations can skip over Packets without knowledge of their payload, enabling straightforward extensibility.

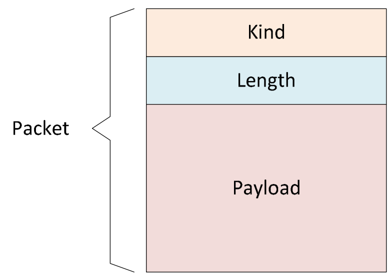


Figure 6.1: Packet

Packets are logically grouped into hierarchical structures. Specifically, as depicted in Figure 6.2, a Bitstream consists of a sequence of Frame structures, each containing the metadata and audio samples required to completely reproduce a Program for a specified interval within its timeline. In particular, each Frame structure contains as complete copy of the Program header information, thereby allowing playback to start on any Frame structure boundary without requiring access to prior or future Frames.

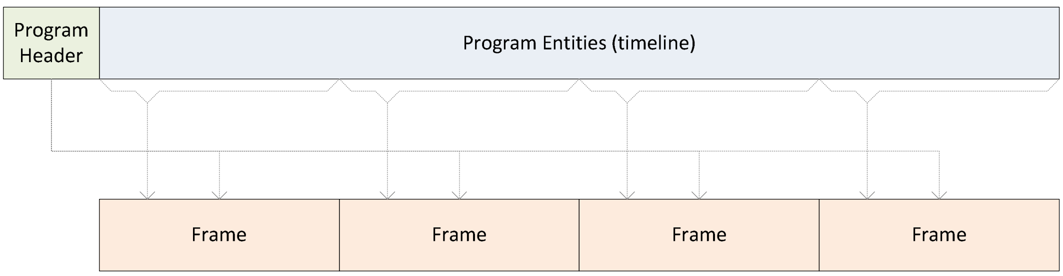


Figure 6.2: Mapping Program to Bitstream Frame Structures

As illustrated in Figure 6.3, Frame structures are further segmented, with Program header information followed by Fragment structures, each containing Entity metadata for a specified time interval within the timeline of the Frame. The audio samples associated with the Frame are contained in a sequence if Asset Frame Packets.

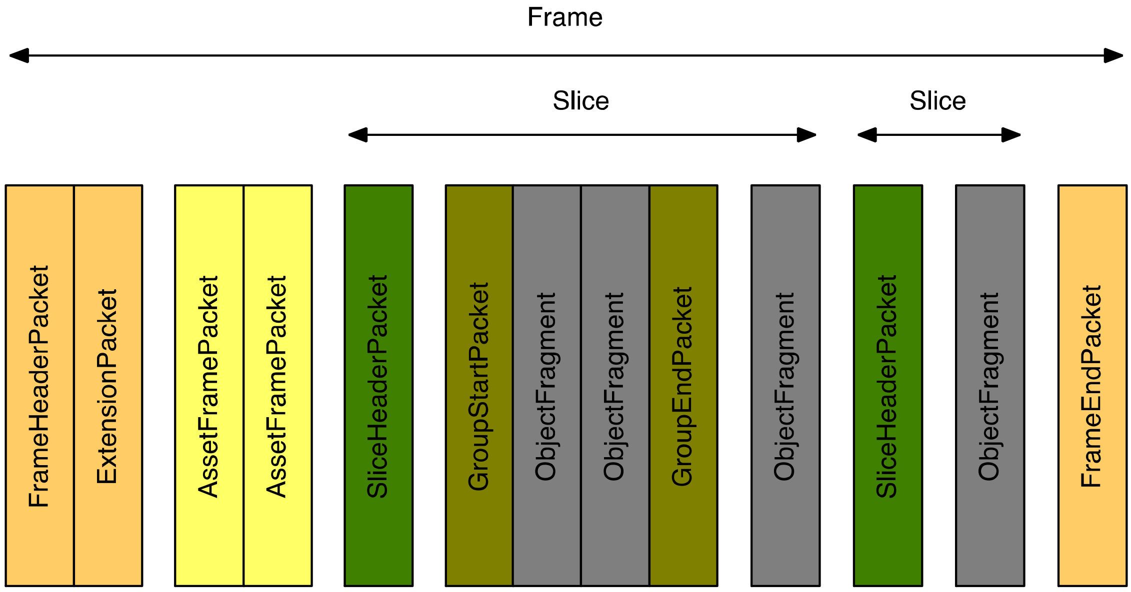


Figure 6.3: Frame Structure

The Program object model makes extensive use of URI as unique identifiers. To reduce minimize overhead, the present document defines mappings between common URI values and shorter Label values.

6.2 Structures

6.2.1 General

The Bitstream syntax is expressed using the operation of a hypothetical parser using the structure specification language described in Annex A.

For extensibility, the Bitstream syntax allows the presence of unknown Packets - captured by fields of type `UnexpectedPacket`. Implementations may safely ignore these unknown Packets.

6.2.2 Bitstream

A Bitstream consists of an ordered sequence of Frames, as specified in Fragment 6.1.

Each Frame is an item of the `fFrame[]` collection. The number of Frames in an MDA Bitstream for a given MDA Program **SHALL** be less than or equal to 2^{64}.

Fragment 6.1: Bitstream Structure

```
aligned struct Bitstream {

    var int i = 0;

    while(!eof) {

        peek PacketHeader fUnknownHeader;

        // Frames

        if (fUnknownHeader.fKind == FrameHeaderPacket.fKind) {
            Frame fFrame[i++];
        }

        // Unknown packets

        else {
            UnexpectedPacket    fUnexpectedPacket;
```

```
        }
    }
}
```

6.2.3 Frame

A Frame **SHALL** correspond to an interval within a Program timeline, with the Program uniquely identified by the `fFrameHeaderPacket.fId` field.

The `MDA::Program.header` object is specified by the `fFrameHeaderPacket` field.

The audio samples required for the reproduction of the Frame **MAY** be contained in the `fAssetFramePacket[]` field.

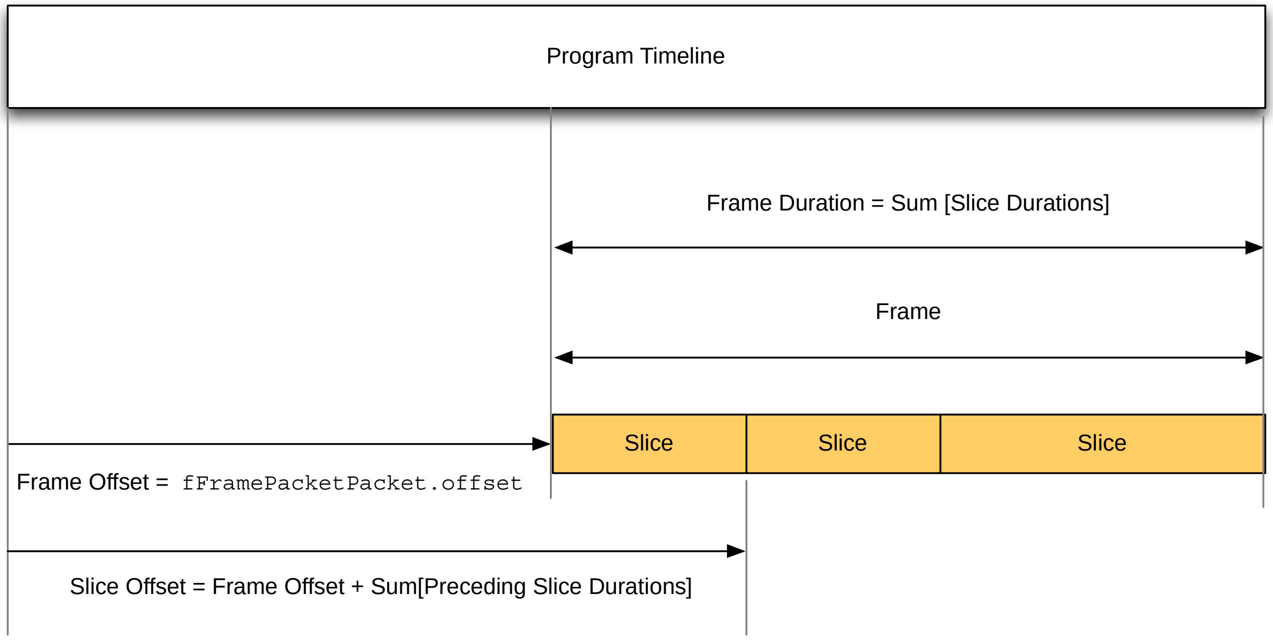


Figure 6.4: Frame and Slice Structure Intervals

The Entity instances associated with the Frame are contained in the `fSlice[]` field.

Each item of `fSlice[]` is a Slice corresponding to a time interval within the time interval of the Frame.

The absolute offset of the Frame timeline within the Program timeline is equal to the value of the `fFrameHeaderPacket.fOffset` field.

The duration of the Frame shall be equal to the sum of the duration of the Slices.

Constraints on Frame duration are not specified by the present document and left to applications.

Two successive Frames do not necessarily belong to the same Program, i.e. a Bitstream may hold multiple Programs.

The total number of Slices in a Frame **SHALL** be less than or equal to 2^{32}.

Fragment 6.2: Frame Structure

```
aligned struct Frame {

    /* process Frame Header packet */

    FrameHeaderPacket fFrameHeaderPacket;

    /* process asset frame packets */

    var int i = 0;

    while(!eof) {
```

```
        peek PacketHeader   fUnknownHeader;

        /* Asset Packets

        if (fUnknownHeader.fKind == AssetFramePacket.fKind) {
            AssetFramePacket fAssetFramePacket[i++];
        }

        // End of assets, start of slices

        else if (fUnknownHeader.fKind == SliceHeaderPacket.fKind) {
            break;
        }

        // Unknown packet

        else {
            UnexpectedPacket fUnexpectedPacket;
        }
    }

    /* process Slices */

    var int j = 0;

    while(!eof) {
        peek PacketHeader   fUnknownHeader;

        // Slices

        if (fUnknownHeader.fKind == SliceHeaderPacket.fKind) {
            Slice   fSlice[j++];

        }

        // End of frame/Slices

        else if (fUnknownHeader.fKind == FrameEndPacket.fKind) {
            break;
        }

        // Unknown packet

        else {
            UnexpectedPacket fUnexpectedPacket;
        }
    }
}
```

6.2.4 Assets

An Asset Frame Packet contains a sequence of file-internal audio samples associated with the Frame. The content of Asset Frame Packet correspond the target of the `assetURI` property in the Core specification. For the present document the `assetURI` for an internal asset shall be of the form "`urn:x-mda:bitstream:afid:<aid>`", where `<aid>` refers to the 32-bit integer identifying the asset. The `AssetFramePacket` type is defined in Clause 6.3.3.

The total number of Asset Frame Packets in Frame **SHALL** be less than or equal to 2^{32}.

6.2.5 Slice Structure

A Slice contains zero or more Entities, contained in the `fEntity[]` field. The number of Entities in a Slice **SHALL** be less than or equal to 2^{32}.

The start offset and duration of each Entity **SHALL** be equal to the start offset and duration of the Slice, respectively.

The duration of the Slice **SHALL** be equal to the `fSliceHeaderPacket.fDuration` field.

The start offset of a Slice **SHALL** be equal to the offset of the Frame plus the sum of the duration of the Slices preceding it.

Fragment 6.3: Slice Structure

```
aligned struct Slice {
    SliceHeaderPacket fSliceHeaderPacket;

// process entities

    var int i = 0;
    while(! eof) {
        peek PacketHeader   fUnknownHeader;

        // SliceHeader, new Slice

        if (fUnknownHeader.fKind == SliceHeaderPacket.fKind) {
            break;
        }

        // FrameEnd

        else if (fUnknownHeader.fKind == FrameEndPacket.fKind) {
            break;
        }
        // Entity

        else if (mIsEntityKind(fUnknownHeader.fKind)) {
            Entity fEntity[i++];
        }

        // Unknown packet

        else {
            UnexpectedPacket        fUnexpectedPacket;
        }
    }
}
```

6.2.6 Entities

The fields of the `Entity` structure **SHALL** be mapped to concrete derived classes of the `MDA::Entity` abstract class.

Table 6.1: Entity Fields to MDA Object Classes

Entity Field	MDA Object Class
Entity.fObjectFragment	MDA::ObjectFragment
Entity.fLFEFragment	MDA::LFEFragment
Entity.fGroup	MDA::Group
Entity.fSwitch	MDA::Switch

Four types are Entities may be present in an MDA bitstream for the present document.

Fragment 6.4: Entity Structure

```
// IsEntity?

#define mIsEntityKind(pKind)
                            (   pKind == ObjectFragmentPacket.fKind     || \
                                pKind == LFEFragmentPacket.fKind    || \
                                pKind == GroupStartPacket.fKind         || \
                                pKind == SwitchStartPacket.fKind)

// Entities

aligned struct Entity {
    peek PacketHeader   fUnknownHeader;

    // ObjectFragment

    if (fUnknownHeader.fKind == ObjectFragmentPacket.fKind) {
        ObjectFragment      fObjectFragment;
    }

    // LFEFragment
```

```
    else if (fUnknownHeader.fKind == LFEFragmentPacket.fKind) {
        LFEFragment          fLFEFragment;
    }

    // Group

    else if (fUnknownHeader.fKind == GroupStartPacket.fKind) {
        Group                       fGroup;
    }

    // Switch

    else if (fUnknownHeader.fKind == SwitchStartPacket.fKind) {
        Switch                  fSwitch;
    }
}
```

6.2.7 LFEFragment

An `LFEFragment` structure corresponds to a single `MDA::LFEFragment` instance.

Fragment 6.5: LFEFragment Structure Syntax

```
aligned struct LFEFragment {
    LFEFragmentPacket   fLFEFragmentPacket;
}
```

The `fLFEFragmentPacket` **SHALL** be mapped to the `MDA::LFEFragment` object.

6.2.8 ObjectFragment

An `ObjectFragment` structure corresponds to a single `MDA::ObjectFragment` instance.

Fragment 6.6: ObjectFragment Structure Syntax

```
aligned struct ObjectFragment {
    ObjectFragmentPacket fObjectFragmentPacket;
}
```

The `fObjectFragmentPacket` shall be mapped to the `MDA::ObjectFragment` object.

6.2.9 Group

A Group structure corresponds to a single `MDA::Group` instance.

Fragment 6.7: Group Entity Structure

```
aligned struct Group {
    GroupStartPacket    fGroupStartPacket;

    // Group Members

    var int i = 0;
    while(! eof) {
        peek PacketHeader   fUnknownHeader;

        // End of Group

        if (fUnknownHeader.fKind == GroupEndPacket.fKind) {
            GroupEndPacket fGroupEndPacket;
            break;
        }

        // Member

        else if (mIsEntityKind(fUnknownHeader.fKind)) {
            Entity fMember[i++];
        }

        // Unknown packet

        else {
            UnexpectedPacket        fUnexpectedPacket;
```

```
        }
    }
}
```

Each member of the `fMember[]` collection shall correspond to a member of the `MDA::Group.members` collection.

6.2.10 Switch

A `Switch` structure corresponds to a single `MDA::Switch` instance.

Fragment 6.8: Switch Entity Syntax

```
aligned struct Switch {
    SwitchStartPacket   fSwitchStartPacket;

    // Members

    var int i = 0;
    while(! eof) {
        peek PacketHeader   fUnknownHeader;

        // End of Switch

        if (fUnknownHeader.fKind == SwitchEndPacket.fKind) {
            SwitchEndPacket fSwitchEndPacket;
            break;
        }

        // Member: there must be at least one (default)

        else if (mIsEntityKind(fUnknownHeader.fKind)) {
            Entity fMember[i++];
        }

        // Unknown packet

        else {
            UnexpectedPacket        fUnexpectedPacket;
        }
    }
}
```

The `fMember[0]` member shall correspond to the `MDA::Switch.default` property and in compliance with Clause 4 is required.

Each remaining member of the `fMember[]` collection shall correspond to a member of the `MDA::Switch.alternates` collection.

6.3 Packets

6.3.1 General

All Label values shall be URI values unless specified otherwise in the present document.

6.3.2 Frame Header Packet

6.3.2.1 General

The `FrameHeaderPacket` signals the start of a Frame and contains information applicable to the Frame and Program as a whole.

For this version of the Bitstream Specification the `MDA::Program.header.constraintSets` property **SHALL** be absent from a `FrameHeaderPacket`.

Fragment 6.9: Frame Header Packet Structure

```
aligned struct FrameHeaderPacket : PacketHeader {
    fKind = kFrameHeaderPacketKind;

    unsigned int(8)                 fBitstreamVersion;
    Label                                   fProgramNamespace;
    UTF8String                          fProgramURI;
    Label                                   fSampleRate;

    // ConstraintSets not supported for BitstreamVersion=3

    OptionalItem<FixedArray<Extension>>     fExtensions;

    // Slice

    PackedUInt64                        fOffset;
    unsigned int(16)                    fDuration;

    OptionalItem<unsigned int(16)>      fCRC;
}
```

6.3.2.2 fBitstreamVersion

The `fBitstreamVersion` field **SHALL** be equal to `kBitstreamVersion=3` for a Bitstream conforming to the present document.

6.3.2.3 fProgramNamespace

For the present document, the `fProgramNamespace` field **SHALL** be equal to `<mdacore>`.

Table 6.2: Program Namespace Labels

Symbol	URI
mdaroot	http://mdaif.org
mdacore	http://mdaif.org/core/1.0

6.3.2.4 fProgramURI

The `fProgramURI` field **SHALL** be equal to `MDA::Program.header.programURI`.

6.3.2.5 fSampleRate

The `fSampleRate` field **SHALL** be equal to `MDA::Program.header.sampleRate.` Implementations **SHALL** support the values `<48KHz>` and `<96KHz>` as listed in Table 6.3.

Table 6.3: Sample Rate Labels

Constant	URI
48 KHz	<mdacore>/labels/sample-rate/48000Hz
96 KHz	<mdacore>/labels/sample-rate/96000Hz

6.3.2.6 fExtensions

Each item of `fExtensions` field **SHALL** correspond to an item of the `MDA::Program.header.extensions` collection. The number of Extensions in a Frame Header Packet **SHALL** be less than or equal to 2^{32}.

6.3.2.7 fOffset

The `fOffset` field **SHALL** be equal to the offset (in samples) of the time interval represented by the Frame within the Program timeline.

6.3.2.8 fDuration

The `fDuration` field **SHALL** be equal to the duration of the time interval represented by the Frame, defined as the sum of all `SliceHeaderPacket.fDuration` values for the Slices within the Frame.

6.3.2.9 fCRC

The `fCRC` field **SHALL** be computed over all fields preceding the `fCRC` field using the 16-bit CRC specified in Figure 6.5, with all bits initialized to 1. The generating polynomial is $x^{16} + x^{12} + x^5 + 1$.

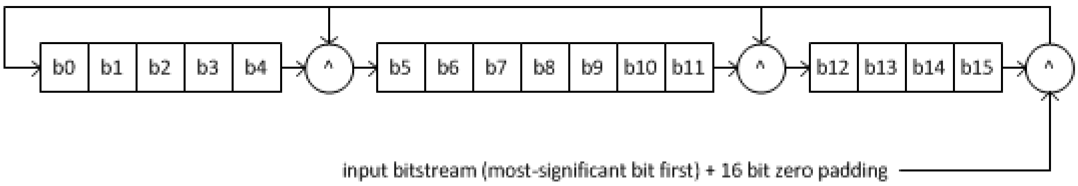


Figure 6.5: CRC_16 Algorithm

The `fCRC` field can be used by implementations to reliably locate the start of a Frame in a Bitstream.

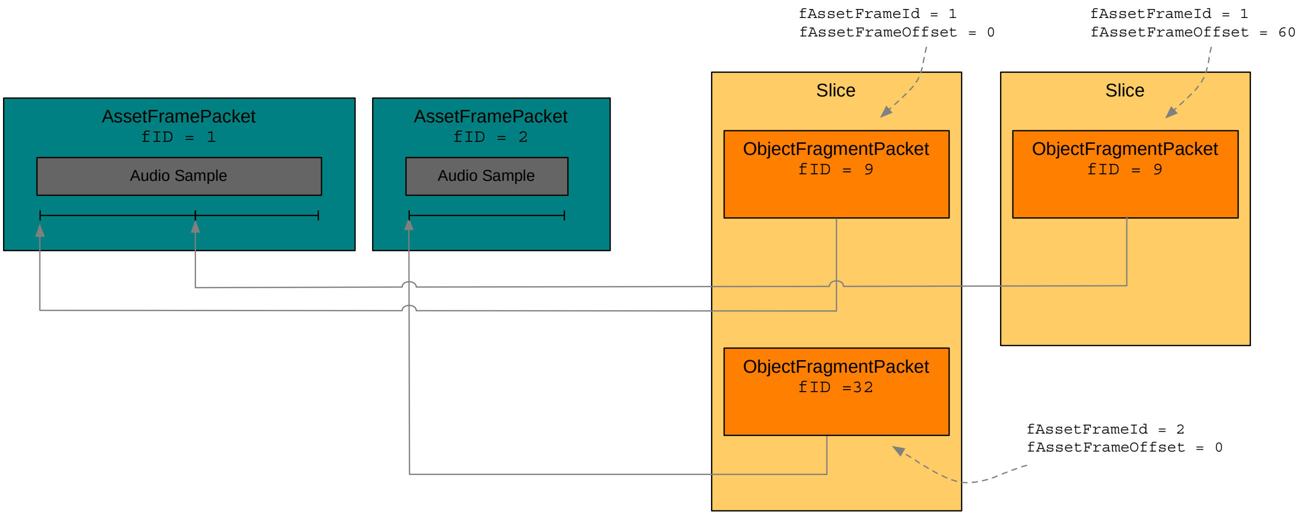


Figure 6.6: Relationship between Asset Frame Packets and Object Fragment Packets

6.3.3 Asset Frame Packet

6.3.3.1 General

Each AssetFramePacket contains a sequence of audio samples associated with the Frame. As illustrated in Figure 6.6, multiple Entities may reference audio samples within the same AssetFramePacket.

Note that an AssetFramePacket does not correspond to MDA core concept.

Fragment 6.10: Asset Frame Packet Structure

```
aligned struct AssetFramePacket : PacketHeader {
    fKind = kAssetFramePacketKind;

    PackedUInt16        fId;
    Label                       fEncoding;
    ByteArray                   fAssetBytes;
}
```

6.3.3.2 fId

The fId field identifies the AssetFramePacket and shall be unique within the Frame.

6.3.4 fAssetEncoding

6.3.4.1 General

The fAssetEncoding field shall identify the format of data contained in the fAssetBytes field.

Implementations **SHALL** support the coding schemes <PCM24> and <PCM32> as listed in Table 6.4.

Table 6.4: Audio Encoding

Constant	URI	Definition
PCM24	<mdacore>/labels/essence-encoding/PCM24	Sequence of 24-bit PCM audio samples, each packed in MSB order.
PCM32	<mdacore>/labels/essence-encoding/PCM32	Sequence of 32-bit PCM audio samples, each packed in MSB order.

6.3.4.2 fAssetBytes

The fAssetBytes field **SHALL** contain a representation of the audio samples. A value of fAssetBytes.fCount equal to 0 shall be interpreted an all 0 (zero) array of audio samples with fAssetBytes.fCount equal to the Frame duration.

6.3.5 Frame End Packet

A FrameEndPacket signals the end of a Frame.

Fragment 6.11: Fragment End Packet Structure

```
aligned struct FrameEndPacket : PacketHeader {
    fKind = kFrameEndPacketKind;
}
```

6.3.6 Slice Header Packet

6.3.6.1 General

A `SliceHeaderPacket` signals the start of a Slice.

Fragment 6.12: Slice Header Packet Structure

```
aligned struct SliceHeaderPacket : PacketHeader {
    fKind = kSliceHeaderPacketKind;

    unsigned int(16)    fDuration;
}
```

6.3.6.2 fDuration

The `fDuration` field **SHALL** be the duration (in samples) of the Slice.

6.3.7 Object Fragment Packet

6.3.7.1 General

An `ObjectFragmentPacket` corresponds to an `MDA::ObjectFragment` instance.

Fragment 6.13: Object Fragment Packet Structure

```
aligned struct ObjectFragmentPacket : MonoSourceFragmentPacket {
    fKind = kObjectFragmentPacketKind;

    OptionalItem<Position>                                  fPosition;
    OptionalItem<unsigned int(8)>                           fAperture;
OptionalItem<unsigned int(8)>                           fDivergence;
    OptionalItem<unsigned int(1)>                           fCoherent;
    OptionalItem<Label>                                         fContentKind;
    OptionalItem<FixedArray<ChannelException>>      fChannelExceptions;
    OptionalItem<FixedArray<PositionException>>     fPositionExceptions;
}
```

6.3.7.2 fPosition

The `fPosition` field **SHALL** be equal to `MDA::ObjectFragment.position`.

6.3.7.3 fAperture

The `fAperture` field **SHALL** be equal to `MDA::ObjectFragment.aperture`. The `fAperture` field is interpreted as `fAperture*(180/255)` in units of degrees. For implementations that use radians, the aperture value shall be interpreted as `fAperture*(π/255)`. In compliance with Clause 4, the default value of `fAperture` **SHALL** be 0.

6.3.7.4 fDivergence

The `fDivergence` field **SHALL** be equal to `MDA::ObjectFragment.divergence`. The `fDivergence` field is interpreted as `fDivergence*(180/255)` in units of degrees. For implementations that use radians, the divergence value shall be interpreted as `fDivergence*(π/255)`. In compliance with Clause 4, the default value of `fDivergence` **SHALL** be 0.

6.3.7.5 fCoherent

The `fCoherent` field **SHALL** be equal to `MDA::ObjectFragment.coherent`.

6.3.7.6 fContentKind

The `fContentKind` field **SHALL** be equal to `MDA::ObjectFragment.contentKind`.

Table 6.5 lists allowed values for `fContentKind`.

Table 6.5: Content Kind Local Labels

Constant	URI
dialog	<mdacore>/labels/content-kind/dialog
effects	<mdacore>/labels/content-kind/effects
music	<mdacore>/labels/content-kind/music

6.3.7.7 fRenderingExceptions

The `fRenderingExceptions` field **SHALL** be equal to `MDA::ObjectFragment.renderingExceptions`.

6.3.8 LFE Fragment Packet

An `LFEFragmentPacket` corresponds to an `MDA::LFEFragment` instance.

Fragment 6.14: LFE Fragment Packet Syntax

```
aligned struct LFEFragmentPacket : MonoSourceFragmentPacket {
    fKind = kLFEFragmentPacketKind;
}
```

6.3.9 Group Start Packet

A `GroupStartPacket` signals the start of a `Group` structure.

Fragment 6.15: GroupStartPacket Syntax

```
aligned struct GroupStartPacket: EntityPacket {
    fKind = kGroupStartPacketKind;
}
```

6.3.10 Group End Packet

A `GroupEndPacket` signals the end of a `Group` structure.

Fragment 6.16: GroupEndPacket Syntax

```
aligned struct GroupEndPacket : PacketHeader {
    fKind = kGroupEndPacketKind;
}
```

6.3.11 SwitchStartPacket

A `SwitchStartPacket` indicates the start of a `Switch` structure.

Fragment 6.17: SwitchStartPacket Syntax

```
aligned struct SwitchStartPacket: EntityPacket {
    fKind = kSwitchStartPacketKind;
}
```

6.3.12 SwitchEndPacket

A `SwitchEndPacket` indicates the end of a `Switch` structure.

Fragment 6.18: SwitchEndPacket Syntax

```
aligned struct SwitchEndPacket : PacketHeader {
    fKind = kSwitchEndPacketKind;
}
```

6.3.13 EntityPacket

6.3.13.1 General

An `EntityPacket` corresponds to the `MDA::Entity` abstract class.

Fragment 6.19: EntityPacket Syntax

```
aligned abstract struct EntityPacket : PacketHeader {
    PackedUInt32                    fId;
    OptionalItem<FixedArray<Extension>> fExtensions;
}
```

6.3.13.2 fId

The `fId` field shall be equal to `MDA::Entity.id`.

6.3.13.3 fExtensions

Each item of `fExtensions` field **SHALL** correspond to an item of the `MDA::Entity.extensions` collection. The number of Extensions in an Entity **SHALL** be less than or equal to 2^{32}.

6.3.14 FragmentPacket

A `FragmentPacket` corresponds to the `MDA::Fragment` abstract class.

The `MDA::Fragment.offset` and `MDA::Fragment.duration` properties **SHALL** be equal to the offset and duration, respectively, of the Bitstream Fragment to which the `FragmentPacket` belongs.

Fragment 6.20: FragmentPacket Syntax

```
aligned abstract struct FragmentPacket : EntityPacket {

}
```

6.3.15 MonoSourceFragmentPacket

6.3.15.1 General

A `MonoSourceFragmentPacket` corresponds to the `MDA::MonoSourceFragment` abstract class.

Fragment 6.21: MonoSourceFragmentPacket Syntax

```
aligned abstract struct MonoSourceFragmentPacket : FragmentPacket {

    UTF8String                                              fAssetURI;
    OptionalItem<unsigned int(16)>          fAssetOffset;
    OptionalItem<unsigned int(5)>               fGain;

}
```

6.3.15.2 fAssetURI

The `MDA::MonoSourceFragment.audioSamples.assetURI` **SHALL** be equal to `fassetURI`.

The `fAssetURI` field for Frame-internal assets **SHALL** be equal to `"urn:x-mda:bitstream:afid:"+AssetFramePacket.fId.toString()`, where `fId` is the `fId` value of the `AssetFramePacket` that contains the audio samples used by this `MonoSourceFragmentPacket`.

A general constraint of `fAssetURIs` for frame Frame-external assets is not provided in the present document and will depend on the application.

6.3.15.3 fAssetOffset

The `MDA::MonoSourceFragment.audioSamples.assetOffset` **SHALL** be equal to `fAssetOffset`.

6.3.15.4 fGain

The `fGain` field **SHALL** be equal to `MDA::MonoSourceFragment.gain`. The interpretation of the `fGain` field is as (`fGain-411)/4` in units of dB. In compliance with Clause 0 the default value of `fGain` **SHALL** be 411. Similarly, the value 0 SHALL be interpreted as `-INF`dB (negative infinity).

6.3.16 UnexpectedPacket

A `UnexpectedPacket` is a Packet with unspecified payload.

Implementations **SHALL** accept `UnexpectedPacket`.

Fragment 6.22: UnexpectedPacket Syntax

```
aligned struct UnexpectedPacket : PacketHeader {

    unsigned int(8) fData[fLength];
}
```

6.4 Common Data Structures

6.4.1 PacketHeader

```
@size(fLength.fValue)
aligned struct PacketHeader {
    Label                           fKind;
    PackedLength        fLength;

    // packet payload follows

}
```

A Packet encapsulates an "opaque" payload, with `fKind` and `fLength` denoting the kind and size of the payload, respectively. The `fLength` value does not include the `fKind` and `fLength` properties (see Annex A). The interpretation (and processing) of the payload is indicated by the `fKind` value.

The `PacketHeader.fKind` field of a `PacketHeader` and any derived type **SHALL** be recorded as a `LocalLabel` for the present document (see Table 6.8).

6.4.2 ChannelGain

6.4.2.1 General

```
struct ChannelGain {

    Label                           fChannel;
    unsigned int(8) fGain;

}
```

Some allowed values for `fChannel` are listed in Table 6.6**.**

Table 6.6: Channel Labels

URI	Symbol
urn:smpte:ul:060E2B34.0401010D.03020101.00000000	L
urn:smpte:ul:060E2B34.0401010D.03020102.00000000	R
urn:smpte:ul:060E2B34.0401010D.03020103.00000000	C
urn:smpte:ul:060E2B34.0401010D.03020104.00000000	LFE
urn:smpte:ul:060E2B34.0401010D.03020105.00000000	Ls
urn:smpte:ul:060E2B34.0401010D.03020106.00000000	Rs
urn:smpte:ul:060E2B34.0401010D.03020107.00000000	Lss
urn:smpte:ul:060E2B34.0401010D.03020108.00000000	Rss
urn:smpte:ul:060E2B34.0401010D.03020109.00000000	Lrs
urn:smpte:ul:060E2B34.0401010D.0302010A.00000000	Rrs

6.4.2.2 fGain

The `fGain` field **SHALL** be equal to `MDA::ChannelGain.gain`. The interpretation of the `fGain` field is as -`fGain/4` in units of dB. In compliance with Clause 4.5.16, the default value of `fGain` is 0.

6.4.3 RenderingException

A `ChannelRenderingException` and `PositionRenderingException` correspond to a `MDA::ChannelRenderingException` and `MDA::PositionRenderingException`, respectively.

```
// Channel Rendering Exception

struct ChannelRenderingException {

    Label       fTargetConfiguration;

    FixedArray<ChannelGain> fGains;

}

// Position Rendering Exception

struct PositionRenderingException {

    Label       fTargetConfiguration;

    Position        fPosition;

}
```

Some allowed target configuration values are listed in Table 6.7.

Table 6.7: Target Configurations Labels

URI	Symbol
urn:smpte:ul:060E2B34.0401010D.03020201.00000000	51
urn:smpte:ul:060E2B34.0401010D.03020202.00000000	71

6.4.4 Labels

```
struct Label {
    const unsigned int(3) kLocalKind = 0;
    const unsigned int(3) kURIKind = 1;

    unsigned int(1)         fIsExtended;
    if (fIsExtended) {
            unsigned int(3)         fKind;
            if (fKind == kURIKind) {
                UTF8String              fURI;
            } else {
                unsigned int(4)         fLength;
                unsigned int(8)         fDigits[fLength + 1];
        }
    } else {
        var fKind = kLocalKind;

        unsigned int(7)         fDigits;
    }
}
```

The `Label` structure can carry either a URI or a sequence of 8-bit unsigned integers, referred to as a `LocalLabel`. A label is a local label if the `fIsExtended` bit is equal to zero with the remaining bits in the byte providing a 7-bit `fDigits` value. It the `fIsExtended` bit is equal to 1, the following 3 bits define the label kind `fKind`. If `fKind` is equal to `kLocalKind` (=0), the following 4 bits hold `fLength`, the number of bytes minus one to represent the `fDigits` sequence. If `fKind` is equal to `kURIKind` (=1), referred to as a `URILabel`, the remainder of the label is a `UTF8String` instance. Note that a label is not guaranteed to be byte aligned, neither is its member `fURI` in case of a `URILabel`.

6.4.5 PackedLength

```
typedef BERUInt32 PackedLength;
```

The `fLength` fields of various MDA structures (e.g. `PacketHeader` and `Extension`) are encoded as `BERUInt32` using the type alias `PackedLength`.

6.4.6 Extension

```
@size(fLength.fValue)
struct Extension {

    Label                   fName;
    Packed                  fLength;

  // Payload follows

}
```

An `Extension` encapsulates an "opaque" payload, with `fName` and `fLength` denoting the kind and size of the payload, respectively. The `fLength` value does not include the `fName` and `fLength` properties. The interpretation (and processing) of the payload is indicated by the `fName` value. Note that an `Extension` is not guaranteed to be byte aligned.

The `Extension.fName` field of an `Extension` and any derived type **SHALL** be recorded as a `URILabel` for the present document.

6.4.7 FixedArray

```
template<class T> struct FixedArray {
    PackedLength    fCount;
    T                                   fItems[fCount];
}
```

A `FixedArray` over `T` holds `fCount` items of type `T`. The value of `fCount` is stored as a `PackedLength` value, followed by `fCount` items of type `T`. Note that a `FixedArray` is not guaranteed to be byte aligned, but its members may be.

6.4.8 Position

6.4.8.1 General

```
struct Position {
    OptionalItem<unsigned int(12)>      fRadius;
    OptionalItem<unsigned int(12)>      fAzimuth;
    OptionalItem<unsigned int(11)>      fElevation;
}
```

6.4.8.2 fRadius

The `fRadius` field ranges over [0,4095] and **SHALL** be interpreted as the rational number `fRadius`/2047. In the D-Cinema application the `fRadius` field **SHALL** be equal to its default value 2 047.

6.4.8.3 fAzimuth

The `fAzimuth` field ranges over [0,4095] and **SHALL** be interpreted as the rational number `(fAzimuth-2048)*(360/2048)` in units of degrees. For implementations that use radians, the azimuth value is interpreted as `(fAzimuth-2048)*(2*π/2048)`. In compliance with Clause 4.5.11.2, the default value of `fAzimuth` **SHALL** be 2 048 (front).

6.4.8.4 fElevation

The `fElevation` field ranges over [0,2047] and shall be interpreted as the rational number `(fElevation - 1023)*(90/1023)` in units of circular degrees. For implementations that use radians, the elevation value is interpreted as `(fElevation-1023)*(π/2046)`. The value 2 047 is reserved and **SHALL NOT** be used.

In compliance with Clause 4.5.11.2, the default value of `fElevation` **SHALL** be 1 023 (zero elevation).

6.4.9 ByteArray

```
typedef FixedArray<unsigned int(8)> ByteArray;
```

A `ByteArray` holds a sequence of bytes. Note that a `ByteArray` is not byte aligned.

6.4.10 BERUInt32

```
struct BERUInt32 {
    unsigned int(1)             fExtended;
    if (fExtended) {
            unsigned int(5)             fReserved;
            unsigned int(2)             fLength;
            unsigned int((fLength + 1) <<  3)   fValue;
    } else {
        unsigned int(7)             fValue;
    }
}
```

The `BERUInt32` data structure encodes a 32-bit integer `fValue`. If `fValue` is smaller than 128, `fValue` is represented by a single byte, with necessarily the most significant bit `fExtended` set to zero. If `fValue` is larger than or equal to 128, the 2-bit field `fLength` hold the number of bytes minus one to represent `fValue` and the `fExtended` bit shall be set equal to 1. For the present document, the `fLength` value **SHALL** be always be set to its maximal value 3, corresponding to 4 bytes to encode `fValue`. Note that `BERUIn32` is not byte aligned.

6.4.11 PackedUInt64

```
struct PackedUInt64 {
    unsigned int(3)             fLength;
    unsigned int((fLength + 1) << 3)    fValue;
}
```

The `fLength` field is equal to the number of bytes minus one to encode `fValue` in Big Endian order. For the present document the `fLength` field **SHALL** be set equal to 7, corresponding to 8 bytes in Big Endian order to encode `fValue`. Note that `PackedUInt64` is not byte aligned.

6.4.12 PackedUInt32

```
struct PackedUInt32 {
    unsigned int(2)             fLength;
    unsigned int((fLength + 1) << 3)    fValue;
}
```

The `fLength` field is equal to the number of bytes minus one to encode `fValue` in Big Endian order. For the present document the `fLength` field **SHALL** be set equal to 3, corresponding to 4 bytes in Big Endian order to encode `fValue`. Note that `PackeUInt32` is not byte aligned.

6.4.13 PackedUInt16

```
struct PackedUInt16 {
    unsigned int(1)             fLength;
    unsigned int((fLength + 1) << 3)    fValue;
}
```

The `fLength` field is equal to the number of bytes minus one to encode `fValue` in Big Endian order. For the present document the `fLength` field **SHALL** be set equal to 1, corresponding to 2 bytes in Big Endian order to encode `fValue`. Note that `PackedUInt16` is not byte aligned.

6.4.14 OptionalItem

```
template<class T> struct OptionalItem {
    unsigned int(1) fPresent;
    if (fPresent) {
        T                               fValue;
    }
}
```

When `fPresent` is `true`, the value of an optional item shall be `fValue`. When `fPresent` is `false`, the value of an optional item shall be t_0 when a default value t_0 for `T` is defined, and `undefined` otherwise and **SHOULD NOT** be relied upon.

6.4.15 UTF8String

```
typedef ByteArray UTF8String;
```

The `UTF8String` data type holds UTF8 encoded character strings [i.1]. Note that an `UTF8String` is not guaranteed to be byte aligned within an MDA bitstream.

6.5 Constants

6.5.1 Packet Kinds

Table 6.8: Packet Kinds

```
const Label kFrameHeaderPacketKind =            {1,0x00,0x01,{0x5A,0xA5}};
const Label kFrameEndPacketKind =               {0,0x01};
const Label kFragmentHeaderPacketKind =         {0,0x02};
const Label kObjectFragmentPacketKind =         {0,0x03};
const Label kAssetFramePacketKind =             {0,0x04};
const Label kGroupStartPacketKind =             {0,0x05};
const Label kGroupEndPacketKind =               {0,0x06};
const Label kSwitchStartPacketKind =            {0,0x07};
const Label kSwitchEndPacketKind =          {0,0x08};
const Label kExtensionsPacketKind =             {0,0x09};
const Label kLFEFragmentPacketKind =            {0,0x0A};
```

6.5.2 Bitstream Version

Table 6.9: Bitstream Version

```
const Label kBitstreamVersion = {0,0x03};
```

7 MDA Broadcast Extensions

7.1 Summary

This clause specifies a broadcast extension the MDA core specification that is backwards compatible with the MDA core specification. The new elements in this extension documents fall in two categories: (i) broadcast specific entities and (ii) broadcast specific metadata. The former are new audible components within an MDA Program, whereas the latter are metadata that are attached to MDA core and broadcast entities as `Extensions` within the `<bpx>` namespace (see Table 7.1). Legacy (non-broadcast) MDA Players will ignore these new elements.

7.2 Higher Order Ambisonics

7.2.1 General

This clause defines the `HOAObjectFragment` as a new broadcast specific entity. The `HOAObjectFragment` class introduces Higher Order Ambisonics (HOA) objects within an MDA stream. HOA objects are not rendered on the MDA reference renderer, but using an HOA renderer. No specific HOA renderer is required by the present document. This new entity is unknown to legacy (non-broadcast) MDA players and will be ignored by such players.

7.2.2 HOAMonoFragment

7.2.2.1 General

An HOAMonoFragment is a named waveform associated with an (Higher Order) Ambisonics sound representation. An HOAMonoFragment **SHALL** be rendered in the context of an HOAObjectFragment (Clause 7.2.3) and **SHALL** be ignored by the normative MDA VBAP renderer.

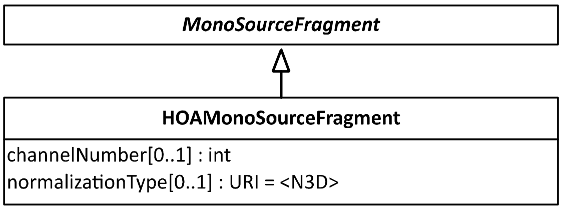


Figure 7.1: HOAMonoSourceFragment

7.2.2.2 Semantics

7.2.2.2.1 channelNumber

The channelNumber property is the ACN order $n*(n+1)+m$, where n and m, $-n \leq m \leq n$ are the order and degree of the Ambisonics component of represented by the HOAMonoFragment [i.3]. If this property is absent, the normalizationType property (if present) **SHALL** be ignored, and any HOAObjectFragment that contains this HOAMonofragment **SHALL** contain an adaptorMatrix element.

7.2.2.2.2 normalizationType

The optional normalizationType property states the normalization scheme for the Legendre functions P_n^m defining the basis functions of Ambisonics decompositions. This property is ignored if the channelNumber property is absent. If absent, N3D normalization <N3D> **SHALL** be assumed [i.3]. The allowed values for normalization are listed in Table 7.5.

7.2.3 HOAObjectFragment

7.2.3.1 General

An HOAObjectFragment aggregates waveforms and metadata for rendering HOA sound fields.

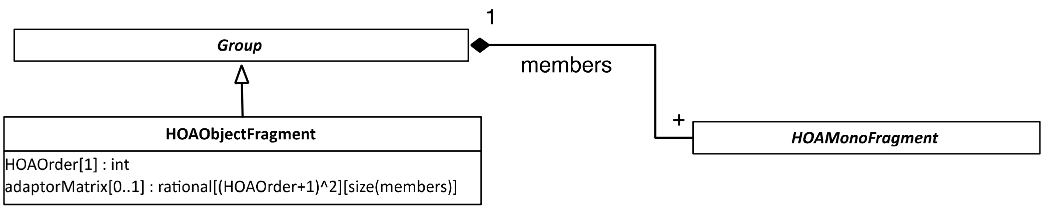


Figure 7.2: HOAObjectFragment

7.2.3.2 Semantics

7.2.3.2.1 HOAOrder

The HOAOrder element indicates the order of the HOAObjectFragment.

7.2.3.2.2 adaptorMatrix

The adaptorMatrix property value is matrix that for converting its member waveforms (in their listed order) to the full component set of an Ambisonics sound field of order HOAOrder. This property **MAY** be absent when implied by the metadata of its HOAMonoFragment members [i.3].

7.2.3.2.3 Members

The `members` property lists the waveform components of the `HOAObjectFragment`.

7.3 Broadcast Extensions

7.3.1 General

This clause defines the MDA broadcast extension set. Each such extension is an instance of the virtual class `Extension`.

7.3.2 BroadcastExtension

7.3.2.1 General

A `BroadcastExtension` is an extensible data structure adding broadcast specific metadata to the core MDA specification. A Broadcast Extension is a virtual class and is the base of a number of concrete broadcast extension subclasses. The `fName` property of a Broadcast Extension shall be in the `<bpxext>` namespace. MDA Players that do not recognize the broadcast extension type may safely ignore such extensions.

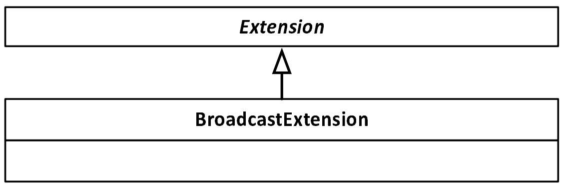


Figure 7.3: Broadcast Extension

7.3.3 Program Broadcast Extension

7.3.3.1 General

A `ProgramBroadcastExtension` is a concrete subclass of the `BroadcastExtension` class and provides metadata for an MDA program. It SHALL be attached to the MDA Program Header for any MDA Program that contains broadcast extension elements. Reversely, the presence of a `ProgramBroadcastExtension` indicates that MDA Program contains broadcast elements.

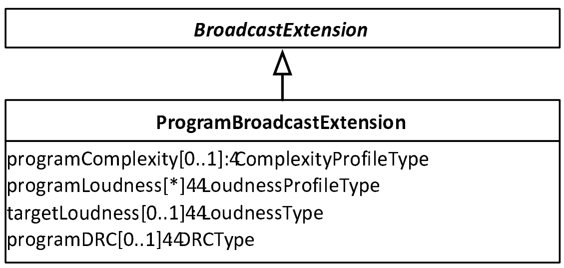


Figure 7.4: Program Extension

7.3.3.2 Semantics

7.3.3.2.1 programComplexity

The `programComplexity` property lists global MDA Program properties that allow an MDA Player to quickly determine if it can render the MDA Program. The structure of `ComplexityProfileType` is defined in Clause 7.4.1.

7.3.3.2.2 programLoudness

The `programLoudness` property indicates the *measured* loudness of the MDA Program. The `LoudnessProfileType` is defined in Clause 7.4.2.

7.3.3.2.3 targetLoudness

Target Loudness Level property indicates the set target Program loudness level of the Program the content creator or programmer wants to achieve once the Program or bitstream is rendered during playback. This metadata value can be used by a downstream loudness monitoring or normalization system as guidance of how much gain to apply or how to adjust the loudness during playback.

Makers of loudness normalization equipment as well as audio compression systems can use this value in conjunction with the other loudness metadata measurements, to maintain proper loudness management and adhere to any possible governing regulations.

The `targetLoudness.value` property **SHALL** be in the range [-35, -4] with `targetLoudness.units` equal to either `<LUFS>` or `<LKFS>`. The `LoudnessType` is defined in Clause 7.4.3.

7.3.3.2.4 programDRC

The `programDRC` property list specific DRC profiles for the MDA Program. The `DRCType` is defined in Clause MDA Reference Renderer.

7.3.3.3 Group Broadcast Extension

7.3.3.3.1 General

The `GroupBroadcastExtension` property provides additional information on the semantics of an MDA Group construct.

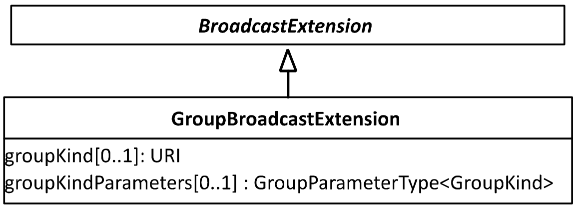


Figure 7.5: Group Extension

7.3.3.4 Semantics

7.3.3.4.1 groupKind

The `groupKind` element indicates an interpretation of the elements of an MDA group. Potential values for this element are provided in Table 7.4.

7.3.3.4.2 groupKindParameters

The `groupKindParameters` element provides additional information associated with `groupKind`. The specifics of this element depend on the value of `groupKind`.

7.3.4 Entity Broadcast Extension

7.3.4.1 General

Extensions to the `Entity` abstract class define extensions that apply to any concrete class derived from `Entity`.

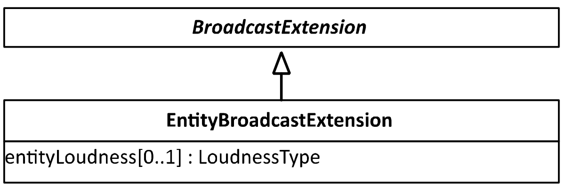


Figure 7.6: EntityBroadcastExtension

7.3.4.2 Semantics

7.3.4.2.1 entityLoudness

The `entityLoudness` property indicates the *measured* loudness of a concrete MDA `Entity` instance. The `LoudnessType` is defined in Clause 7.4.2.

7.3.5 ObjectFragment Broadcast Extension

7.3.5.1 General

The `ObjectFragmentBroadcastExtension` class defines metadata pertaining to broadcast specific processing and rendering of `ObjectFragment` instances.

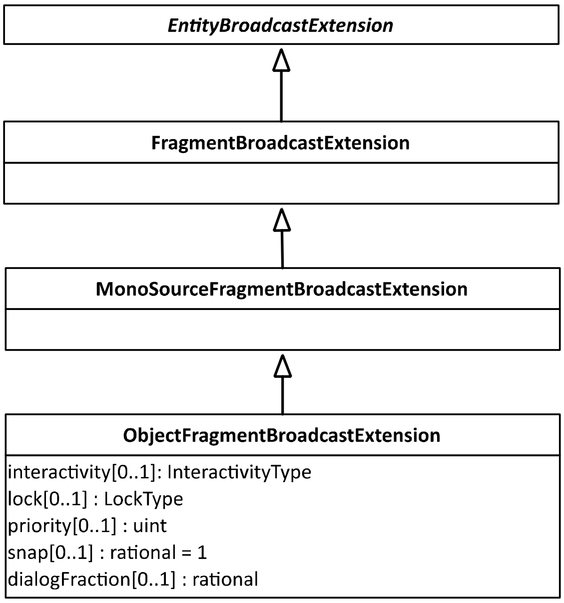


Figure 7.7: ObjectFragment Broadcast Extension

7.3.5.2 Semantics

7.3.5.2.1 Interactivity

The `interactivity` element indicates which core metadata elements in the `ObjectFragment` is authored for interactivity and MAY be modified by subsequent processing or rendering and how. The `InteractivityType` data type is defined in Clause 7.4.6.

A compliant MDA processor/renderer SHOULD not modify any core metadata that are not explicitly marked as interactive. An `ObjectFragment` with an Interactivity element SHALL not include any `RendererException`.

7.3.5.2.2 Lock

The `lock` element indicates whether or not the object under consideration is available for processing and/or rendering. An MDA compliant processor or player SHOULD not attempt process a locked object when it is does not have the appropriate unlock key: processing or playing a locked object will give unpredictable results. The `LockType` data type is defined in Clause 7.4.7.

7.3.5.2.3 priority

The `priority` property indicates the priority for various components in an MDA Program stream and may be used to guide down stream processing decisions. For example, as an MDA Programs goes through a distribution chain, it may encounter scenarios where a reduction of objects in the program needs to occur (e.g. to meet certain delivery profiles or bandwidth restrictions). Object priority parameters will act a communication pathway between the content creator and the encoding engine, proving guidance on what objects are deemed more important for preservation when full object preservation is not viable. Objects with higher priority SHOULD be preserved over objects with lower priority.

The number 0 indicates lowest priority with increasing priority indicated by higher numbers.

7.3.5.2.4 snap

The `snap` property, if present, states that as a post-processing phase after VBAP rendering, the largest gain value larger than then the value of `snap` **SHALL** be set to 1 and all other gain values **SHALL** be set to 0. Effectively, the object is snapped to the nearest speaker if the object is (spatially) close enough (as determined by the value of `snap`). The absence of `snap` (i.e. no snapping) is equivalent to the presence of snap with default value equal to 1.

The value of snap **SHALL** be between 0 and 1.

7.3.5.2.5 dialogFraction

The `dialogFraction` property, if present, states the fraction of the audio essence considered to be of dialog kind. The `dialogFraction` ranges from 0 to 1 in increments of 1/127. A value of 0 corresponds to a Fragment without dialog, whereas a value of 1 correspond a Fragment that is pure dialog.

The value of this property is to be used as a pre-factor of the `dialogGain` property as defined in Clause 7.4.4.2.2.

7.4 Data Types

7.4.1 ComplexityProfileType

7.4.1.1 General

The `ComplexityProfileType` data type aggregates global parameters that constrain the complexity of an MDA Program and allow an MDA Player to quickly determine whether or not a Program can be played.

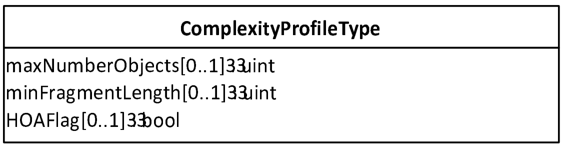


Figure 7.8: ComplexityProfileType

7.4.1.2 Semantics

7.4.1.2.1 maxNumberObjects

The `maxNumberObjects` element defines the maximum number of simultaneous *audible* objects that can exist at any given moment in time.

7.4.1.2.2 minFragmentLength

The `minFragmentLength` element defines the minimum duration of a fragment in terms of MDA Program clock ticks.

7.4.1.2.3 HOAFlag

The `HOAFlag` property indicates whether the MDA Program contains HOA objects (`true`) or not (`false`).

7.4.2 LoudnessProfileType

7.4.2.1 General

The `LoudnessProfileType` data type is a multi-component data structure listing various measured loudness properties of all audible components of the object under consideration. It may be applied to:

1) `Entity`: the measured loudness of the aggregated and rendered `audioEssence` properties (recursively) of the MDA Entity at the center of the MDA unit sphere. For example, for an `ObjectFragment` it measures the loudness of `audioEssence` of as played out of the MDA Reference Rendered. For a Group Entity, it measures the loudness of all members of the Group Entity played out simultaneously.

2) `Program`: the measured loudness of the MDA program as a whole.

Loudness measurements are with reference to an MDA Reference Renderer for a given `measurementConfiguration`, with the listener in the center of the MDA unit sphere.

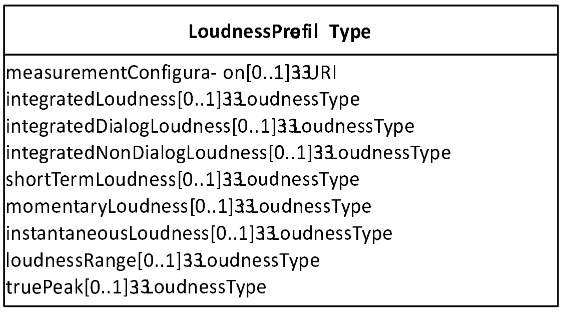


Figure 7.9: LoudnessProfileType

7.4.2.2 Semantics

7.4.2.2.1 measurementConfiguration

The `measurementConfiguration` property states the playback context of an MDA Reference Renderer in the context of which the loudness measurements were performed (see Clause 5.2).

7.4.2.2.2 IntegratedLoudness

The `IntegratedLoudness` property is the long-term integrated loudness of the object under consideration according to [2] and [3].

The `IntegratedLoudness.value` values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1 in units of either `<LUFS>` or `<LKFS>`.

7.4.2.2.3 IntegratedDialogLoudness

The `IntegratedDialogLoudness` property is the long-term loudness of the dialog components only of the object under consideration according to [2] and [3].

The `IntegratedDialogLoudness.value` values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1) in units of either `<LUFS>` or `<LKFS>`.

7.4.2.2.4 IntegratedNonDialogLoudness

The `IntegratedNonDialogLoudness` property is the long-term loudness of the non-dialog components only of the object under consideration according to [2] and [3].

The `IntegratedNonDialogLoudness.value` values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1) in units of either `<LUFS>` or `<LKFS>`.

7.4.2.2.5 ShortTermLoudness

The ShortTermLoudness property is the short-term loudness of the object under consideration according to [2] and [3].

The ShortTermLoudness.value values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1) in units of either <LUFS> or <LKFS>.

7.4.2.2.6 MomentaryLoudness

The MomentaryLoudness property is the momentary loudness of the object under consideration according to [2] and [3]. This property **SHALL** only apply to an MDA entity within the duration of the MDA Entity.

The MomentaryLoudness.value values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1) in units of either <LUFS> or <LKFS>.

7.4.2.2.7 InstantanenousLoudness

The InstantaneousLoudness property is the ultra-short loudness of the object under consideration. This property is useful for the computation of Dynamic Range Control curves. This measure is similar to ShortTermLoudness, differing only in the measurement duration being equal to 5 ms, only including the samples that are completed included in the 5 ms interval.

The InstantenousLoudness.value values **SHALL** be in the range [-700,100]/10 (i.e. -70 to 10 with a resolution of 0.1) in units of either <LUFS> or <LKFS>.

7.4.2.2.8 LoudnessRange

The LoudnessRange property is the loudness range of the object under consideration according to [5].

The LoudnessRange.value values **SHALL** be in the range [0,800]/10 (i.e. 0 to 80 with a resolution of 0.1) in units of <LU>.

7.4.2.2.9 TruePeak

The TruePeak property is the true-peak value of the object under consideration according to [2] and [3].

The TruePeak.value values **SHALL** be in the range [-700,100]/10 in units of <dbTP> with a resolution of 0.1.

7.4.3 LoudnessType

7.4.3.1 General

Loudness is measured as a rational value associated with a particular unit of loudness.

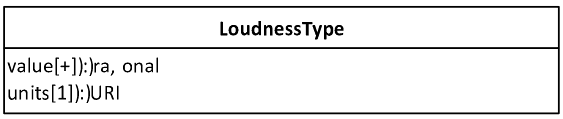


Figure 7.10: LoudnessType

7.4.3.2 Semantics

7.4.3.2.1 value

The value property provides an array of numerical loudness values, with units given in the units property. The loudness values are uniformly distributed over the duration of the object for which loudness values are provided. The range of allowed values depends on application context.

7.4.3.2.2 units

The units property indicates the unit of loudness associated with the value property. The present document allows four loudness related units, as listed in Table 7.3.

7.4.4 DRCType

7.4.4.1 General

Dynamic Range Control profiles are defined with unique compression curves appropriate for various types of content and contexts. Content creators should be familiar with several content-specific profiles such as: Music Light, Music Standard, Film Light, Film Standard and Speech, as they have been included in legacy systems. Additional object-specific or context-specific profiles may also be included (e.g. Special effects objects, Dialog objects, Night-mode context).

It may be desirable to modify the gain of dialog elements independently from DRC profile characteristics. The `dialogGain` data type indicates the amount of gain to apply to dialog elements and may be unique to the content or context.

DRCType
DRCProfile[*] : DRCCurve dialogGain[0..1] : rational

DRCCurve
profileURI : URI curve[0..1]: Curve

Curve
units : URI points[*] : Pair<int,int>

Figure 7.11: DRCType

7.4.4.2 Semantics

7.4.4.2.1 DRCProfile

The `DRCProfile` property defines the input/output compression curves that may be used to generate DRC gain sequences to be applied during the consumer decode stage.

A `DRCProfile` consists of a `profileURI`, identifying the `DRCProfile`, and optionally a compression curve, specified as set of input/output points, defining a linearly interpolated curve between (-61,-61) and (0,0), with units of `<LUFS>` or `<LKFS>` for point coordinates. If the compression curve is absent, the `profileURI` **SHALL** refer to a standard `DRCProfile` with known compression curve.

7.4.4.2.2 DialogGain

The `dialogGain` property specifies the amount of gain to be applied to dialog elements and is units of `dB`.

7.4.5 GroupParameterType<kPBXGroupKindBED>

7.4.5.1 General

A channel-based bed is modeled as an MDA group object of kind `kPBXGroupKindBed`. The type `GroupParameterType<kPBXGroupKindBed>` lists the parameters that apply to `kPBXGroupKindBed`.

GroupParameterType<kPBXGroupKindBed>
renderConfiguration[0..1] : URI

Figure 7.12: Bed Parameters

7.4.5.2 Semantics

7.4.5.2.1 configuration

The `renderConfiguration` element indicates the channel layout of the bed. The value of this element is a standard URI in an appropriate namespace. The (broadcast extension) `members` property of the MDA group SHALL match the (standard) definition of the `renderConfiguration` element. For example, a 5.1 render configuration consists of a single `LFEFragment` - rendered to the `LFESpeaker` - and 5 `ObjectFragments`, rendered by RendererException to L, R, C, Rs and Ls.

7.4.6 InteractivityType

7.4.6.1 General

The `InteractivityType` data type indicates which metadata are authored for interactivity, are eligible for modification and to what degree. A compliant MDA processor/renderer SHOULD not modify any metadata that are not explicitly marked as interactive.

InteractivityType
azimuthDelta[0..1] : rational = 0 elevationDelta[0..1] : rational = 0 apertureDelta[0..1] : rational = 0 divergenceDelta[0..1] : rational = 0 gainDelta[0..1] : rational = 0

Figure 7.13: InteractivityType

7.4.6.2 Semantics

7.4.6.2.1 azimuthDelta

The `azimuthDelta` property specifies the allowed delta for Object azimuth. The value `azimuthDelta` denotes the allowed delta for Object azimuth.

7.4.6.2.2 elevationDelta

The `elevationDelta` property specifies the allowed delta for Object elevation. The value `elevationDelta` denotes the allowed delta for Object elevation.

7.4.6.2.3 apertureDelta

The `apertureDelta` property defines the allowed variation for Object aperture. The value `apertureDelta` denotes the allowed delta for Object aperture.

7.4.6.2.4 divergenceDelta

The `divergenceDelta` property defines the allowed variation for Object divergence. The value `divergenceDelta` denotes the allowed delta for Object divergence.

7.4.6.2.5 gainDelta

The `gainDelta` property defines the allowed variation for Object gain, controlling the power of the audio essence for the object. The value `gainDelta` denotes the allowed delta for Object gain.

7.4.7 LockType

7.4.7.1 General

The `LockType` data type specifies necessary information for unlocking the waveform(s) associated with an object. `LockType` parameters will implicitly apply to all children of an MDA object, unless overridden by an explicit child `LockType` parameter.

LockType
locker[0..1],:,URI keyID[0..1],:,anyType

Figure 7.14: LockType

7.4.7.2 Semantics

7.4.7.2.1 locker

The `Locker` property specifies the method used for obfuscating the object waveform. The Locker property is specified by a standardized URI.

7.4.7.2.2 keyID

The `KeyID` property specifies a pointer to a Locker-specific unlock key. Access to the unlock key is controlled by a DRM context that is out of scope for MDA. The `KeyID` property is opaque to MDA and specific to the Locker property.

7.5 Constants

7.5.1 Conventions

7.5.1.1 Namespaces

Constants defined in the present document are defined in the context of one of two namespaces (see Table 7.1). A namespace "ns" is included in other definitions using the prefix notation "<ns>/".

Table 7.1: Broadcast Extension Namespaces

Namespace	URI	Definition
bpx	<mdaroot>/bpx/1.0	MDA BPX Namespace
bpxext	<bpx>/extension	BPX Extension Namespace

7.5.1.2 Constants

In the tables below, every constant is defined both with a short name and a fully qualified name. The fully qualified name is obtained from the short name by appending it to a constant-class specific prefix. When the context is clear, the short symbol name encapsulated with '<' and '>' MAY be used. Otherwise the fully qualified name SHALL be used.

7.5.2 Profile Constants

Implicit prefix: `kPBXProfile`.

Table 7.2: Profile Constants

Short Symbol	URI	Definition
BASELINE	<bpx>/profile/low	maxNumberObjects = 4
MAIN	<bpx>/profile/medium	maxNumberObjects = 16
HIGH	<bpx>/profile/high	maxNumberObjects = 26

7.5.3 loudnessUnit Constants

Implicit prefix: `kPBXLoudnessUnit`.

Table 7.3: Loudness Related Units

Short Symbol	URI	Definition
LUFS	<bpx>/loudnessunit/LUFS	This constant indicates the LUFS loudness unit as defined in [3] LUFS is an absolute measure, with one unit of LUFS corresponding to 1 dB.
LKFS	<bpx>/loudnessunit/LKFS	This constant indicates the LKFS loudness unit as defined in [2]. LKFS is an absolute measure with one unit of LKFS corresponding to 1 dB.
dBTP	<bpx>/loudnessunit/dBTP	This constant indicates the dBTP unit as defined in [2]. dBTP is an absolute measure with one unit of dBTP corresponding to 1 dB.
LU	<bpx>/loudnessunit/LU	This constant indicates the LU loudness unit as defined in [3]. LU is a relative measure with one unit of LU corresponding to 1 dB.

7.5.4 groupKind Constants

Implicit prefix: `kPBXGroupKind`.

Table 7.4: Group Kinds

Short Symbol	URI	Definition
BED	<bpx>/groupkind/bed	This constant indicates that the MDA group models a channel-based bed. The groupKind property MAY be associated with a groupKindParameterType element in the encapsulating extension class.

7.5.5 HOA Normalization

Implicit prefix: `kPBXHOANormalization`.

Table 7.5: HOA Normalization

Short Symbol	URI	Definition
FuMa	<bpx>/hoa/normalization/fuma	Furse-Malhalm normalization, only applicable for HOA order ≤ 3.
N3D	<bpx>/hoa/normalization/n3d	N3D normalization
SN3D	<bpx>/hoa/normalization/sn3d	SN3D normalization
N2D	<bpx>/hoa/normalization/n2d	N2D normalization
SN2D	<bpx>/hoa/normalization/sn2d	SN2D normalization

7.6 BPX Bitstream

7.6.1 General

The following clauses describe the broadcast elements in the broadcast extended MDA bitstream.

7.6.2 Higher Order Ambisonics

7.6.2.1 HOAMonoSourceFragment

The bitstream `HOAMonoSourceFragmentPacket` implements the metadata model `HOAMonoSourceFragment` class of Clause 7.2.1.

Fragment 7.1: HOAMonoSourceFragment

```
aligned struct HOAMonoSourceFragmentPacket : MonoSourceFragmentPacket {

    fKind = kHOAMonoSourceFragmentPacketKind;

    OptionalItem<int(16)>                  channelNumber;
    OptionalItem<UTF8String>               normalizationType;

}
```

7.6.2.2 HOAObjectFragment

The `HOAObjectFragmentStartPacket` and `HOAObjectFragmentEndPacket` implement the metadata model `HOAObjectFragment` class of Clause 7.2.3. An element of the `adaptorMatrix` field **SHALL** be interpreted as divided by 1 024 (i.e. the 10 rightmost LSB bits are fractional). The `members` of an `HOAObjectFragment` are recorded between the Start and End packet, as in the case of a MDA Core Group.

Fragment 7.2: HOAObjectFragment

```
aligned struct HOAObjectFragmentStartPacket : GroupPacket {

    fKind = kHOAObjectFragmentPacketKind;

    int(8)      HOAOrder;
    OptionalItem<int(16)[(HOAOrder+1)^2][size(members)]>    adaptorMatrix;

}

aligned struct HOAObjectFragmentEndPacket : PacketHeader {

    fKind = kHOAObjectFragmentEndPacketKind;

}
```

7.6.3 Broadcast Extensions

7.6.3.1 General

The bitstream `BroadcastExtension` structure implements the metadata model `BroadcastExtension` class of Clause 7.3.1. The `length` field **SHALL** be equal to the length (in bytes) of the remaining data in a concrete instance of the `BroadcastExtension` class. This will allow applications that do not recognize Broadcast Extensions to quickly skip such extension.

Fragment 7.3: BroadcastExtension

```
aligned struct BroadcastExtension {

    Label                       fName = <bpxext>/…;
    PackedLength    length;
}
```

7.6.3.2 ProgramBroadcastExtension

The bitstream `ProgramBroadcastExtension` implements the metadata model `ProgramBroadcastExtension` class of Clause 7.3.3.

Fragment 7.4: ProgramBroadcastExtension

```
aligned struct ProgramBroadcastExtension : BroadcastExtension {

  fName = <bpxext>/program-extension;

  OptionalItem<ComplexityProfileType>   programComplexity;
  FixedArray<LoudnessProfileType>           programLoudness;
  OptionalItem<LoudnessType>                    targetLoudness;
  OptionalItem<DRCType>                         programDRC;

}
```

7.6.3.3 Group Broadcast Extension

The bitstream `GroupBroadcastExtension` implements the metadata model `GroupBroadcastExtension` class of Clause 7.3.3.3.

Fragment 7.5: GroupBroadcastExtension

```
aligned struct GroupBroadcastExtension : BroadcastExtension {

  fName = <bpxext>/group-extension;

  OptionalItem<UTF8String>                           groupKind;
  OptionalItem<GroupParameterKind<groupKind>>        groupKindParameters;
}
```

7.6.3.4 Entity Broadcast Extension

The bitstream `EntityBroadcastExtension` implements the metadata model `EntityBroadcastExtension` class of Clause 7.3.4.

Fragment 7.6: EntityBroadcastExtension

```
aligned struct EntityBroadcastExtension : BroadcastExtension {

  fName = <bpxext>/entity-extension;

  OptionalItem<LoudnessType>                entityLoudness;

}
```

7.6.3.5 ObjectFragment Broadcast Extension

The bitstream `ObjectFragmentBroadcastExtension` implements the metadata model `ObjectFragmentBroadcastExtension` class of Clause 7.3.5. The `snap` value shall be interpreted with the rightmost 7 LSB bits as fractional.

```
aligned struct ObjectFragmentBroadcastExtension :
                                        MonoSourceFragmentBroadcastExtension {

  fName = <bpxext>/object-extension;

  OptionalItem<InteractivityType>       interactivity;
  OptionalItem<LockType>                    lock;
  OptionalItem<unint(8)>                    priority;
  OptionalItem<uint(8)>                     snap;
  OptionalItem<uint(8)>                     dialogFraction;

}
```

7.6.4 Data Types

7.6.4.1 General

The clauses below specify the bitstream data structures for the data types models in Clause 7.4.

7.6.4.2 ComplexityProfileType

The bitstream `ComplexityProfileType` structure implements the metadata model `ComplexityProfileType` class of Clause 7.4.1.

Fragment 7.7: ComplexityProfileType

```
aligned struct ComplexityProfileType {

  OptionalItem<uint(8)>             maxNumberObjects;
  OptionalItem<uint(8)>             minFragmentLenght;
  OptionalItem<uint(1)>             HOAFlag;

}
```

7.6.4.3 LoudnessProfileType

The bitstream `LoudnessProfileType` structure implements the metadata model `LoudnessProfileType` class of Clause 7.4.2.

Fragment 7.8: LoudnessProfileType

```
aligned struct LoudnessProfileType {

  OptionalItem<UTF8String>            measurementConfiguration;
  OptionalItem<LoudnessType>          integratedLoudness;
  OptionalItem<LoudnessType>          integratedDialogLoudness;
  OptionalItem<LoudnessType>          integratedDialogNonLoudness;
  OptionalItem<LoudnessType>          shortTermLoudness;
  OptionalItem<LoudnessType>          momentaryLoudness;
  OptionalItem<LoudnessType>          loudnessRange;
  OptionalItem<LoudnessType>          truePeak;

}
```

7.6.4.4 LoudnessType

The bitstream `LoudnessType` structure implements the metadata model `LoudnessType` class of Clause 7.4.3. The `value` property specifies the loudness value with a resoultion of 0.1 units.

Fragment 7.9: LoudnessType

```
aligned struct LoudnessType {

  FixedArray<int(10>        value;
  UTF8String                units;

}
```

7.6.4.5 DRCType

The bitstream `DRCType` structure implements the metadata model `DRCType` class of Clause 7.4.4. The `dialGain` value is in units of dB, with the two rightmost LSB bits interpreted as fractional.

The coordinates of a `Curve` point as read from the bitstream **SHALL** be negated before interpretation as compression curve point.

Fragment 7.10: DRCType

```
aligned struct DRCType {

  FixedArray<DRCCurve>              DRCProfiles;
  OptionalItem<int(8)>              dialogGain;

}

aligned struct DRCCurve {

  UTF8String                                profileURI;
  OptionalItem<Curve>                   curve;

}

aligned struct Curve {

  UTF8String                                        units;
  FixedArray<Pair<uint(8),uint(8)>>     curve;

}
```

7.6.4.6 GroupParameterType<BED>

The bitstream `GroupParameterType<BED>` structure implements the metadata model `GroupParameterType<BED>` class of Clause 7.4.5.

Fragment 7.11: GroupParameterType<BED>

```
aligned struct GroupParameterType<BED> {

  OptionalItem<UTF8String>          renderConfiguration;
}
```

7.6.4.7 InteractivityType

The bitstream `InteractivityType` structure implements the metadata model `InteractivityType` class of Clause 7.4.6. The positional and extent degrees of freedom are measures in units of degrees. The `gainDelta` value is in units of dB, with the two rightmost LSB bits of `gainDelta` interpreted as fractional.

Fragment 7.12: InteractivityType

```
aligned struct InteractivityType {

  OptionalItem<uint(8)>                 azimuthDelta;
  OptionalItem<uint(8)>                 elevationDelta;
  OptionalItem<uint(8)>                 apertureDelta;
  OptionalItem<uint(8)>                 divergenceDelta;
  OptionalItem<uint(8)>                 gainDelta;

}
```

7.6.4.8 LockType

The bitstream `LockType` structure implements the metadata model `LockType` class of Clause 7.4.7.

Fragment 7.13: LockType

```
aligned struct LockType {

  OptionalItem<UTF8String>          locker;
  OptionalItem<ByteArray>           keyID;
}
```

Annex A (normative): Structured Specification Language

A.1 General

The structure specification language used in the present document adopts a C-like syntax and is not unlike that used in MPEG standards. All standard integer mathematical operators are supported.

A.2 Macro

A macro is declared as follows:

```
#define MacroName(Argument1, Argument2,…, ArgumentN) MacroBody
```

Each occurrence of `MacroName(Value1, Value2,…, ValueN)` is replaced by `MacroBody` with `Argument1, Argument2,…, ArgumentN` substituted for `Value1, Value2,…, ValueN`.

A.3 Structure

A structure is declared as follows:

```
[@size(Value)]
[aligned] [template<TemplateParameters>] struct NameOfStructure {}
```

The optional `aligned` keyword indicates that the structure is aligned to the next byte boundary in the bitstream, instead of to the next bit boundary.

The optional `@size()` annotation explicitly specifies the *additional* size of the structure, adding to the size implied by the explicitly listed fields of the structure.

The optional `template<>` allows the structure definition to be parameterized.

A structure is always read most significant bit first.

A structure, as well as any field declared within are bit-packed unless specified otherwise, e.g. using the `aligned` keyword.

A.4 Basic Type

The following signed and unsigned integer types are defined:

```
unsigned int(NumberOfBits)
int(NumberOfBits)
```

A.5 Type Aliasing

A structure or basic type can be aliased, i.e. referred to using a different name, as follows:

```
typedef NameOfType NameOfAliasedType;
```

A.6 Control Statements

The following control statements are supported:

```
if (ConditionIsTrue) then {} else if {} else {}
while (ConditionIsTrue) {}
do {} while (ConditionIsTrue);
for (PreLoopStatement; ConditionIsTrue; LoopTailStatement) {}
```

A control statement can be escaped using the `break` keyword, or its next iteration started with the `continue` keyword.

A.7 Fields

A Field read from the Bitstream is declared as follows:

```
[peek][aligned] TypeOfField[<TemplateParameters>] NameOfField[[Cardinality]];
```

The `peek` keyword indicates that the read pointer within the stream should not be incremented when the field is read.

The `aligned` keyword indicates that the field is aligned to the next byte boundary in the Bitstream, instead of the next bit boundary.

The value of a Field can be set as follows:

```
NameOfField = Value;
```

Variables

A Variable is declared as follows:

```
var TypeOfVariable NameOfVariable[[Cardinality]] [= InitializationValue];
```

A Variable is never read from the stream.

A.8 Constants

A Constant is declared as follows:

```
const TypeOfConstant NameOfConstant[[Cardinality]] [= InitializationValue];
```

A Constant is never read from the stream and its value cannot change.

Annex B (informative): XML MDA Broadcast Schema

The code fragment below provides an XML definition of the MDA Broadcast Metadata within the namespace `http://mdaif.org/bpx/1.0`.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema targetNamespace="http://mdaif.org/bpx/1.0/"
    xmlns:bpx="http://mdaif.org/bpx/1.0/"
    xmlns:mda="http://mdaif.org/core/1.0/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

    <xs:complexType name="HOAMonoSourceFragment">
        <xs:sequence>
            <xs:element name="hoaMonoSourceGragment" type="mda:MonoSourceFragment"/>
            <xs:element name="channelNumber" type="xs:nonNegativeInteger" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="normalizationType" type="mda:URI" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="HOAObjectFragment">
        <xs:sequence>
            <xs:element name="group" type="mda:Group"/>
            <xs:element name="HOAOrder" type="xs:nonNegativeInteger"/>
            <xs:element name="adapterMatrix" type="mda:MatrixType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ProgramBroadCastExtension">
        <xs:sequence>
            <xs:element name="extensionURI" type="mda:URI"
fixed="http://mdaif.org/bpx/1.0/extension/program-extension"/>
            <xs:element name="programComplexity" type="bpx:ComplexityProfileType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="programLoudness" type="bpx:LoudnessProfileType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="targetLoudness" type="bpx:LoudnessType" minOccurs="0" maxOccurs="1"/>
            <xs:element name="programDRC" type="bpx:DRCType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="EntityBroadcastExtension">
        <xs:sequence>
            <xs:element name="extensionURI" type="mda:URI"
fixed="http://mdaif.org/bpx/1.0/extension/entity-extension"/>
            <xs:element name="entityLoudness" type="bpx:LoudnessType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ObjectFragmentBroadcastExtension">
        <xs:sequence>
            <xs:element name="extensionURI" type="mda:URI"
fixed="http://mdaif.org/bpx/1.0/extension/object-extension"/>
            <xs:element name="interactivity" type="bpx:InteractivityType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="lock" type="bpx:LockType" minOccurs="0" maxOccurs="1"/>
            <xs:element name="priority" type="xs:nonNegativeInteger" minOccurs="0" maxOccurs="1"/>
            <xs:element name="snap" type="mda:Rational" minOccurs="0" maxOccurs="1"/>
            <xs:element name="dialogFraction" type="mda:Rational" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ComplexityProfileType">
        <xs:sequence>
            <xs:element name="maxNumberObjects" type="xs:nonNegativeInteger" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="minFragmentLenght" type="xs:nonNegativeInteger" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="HOAFlag" type="xs:boolean" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
```

```
    <xs:complexType name="LoudnessProfileType">
        <xs:sequence>
            <xs:element name="measurementConfiguration" type="mda:URI" minOccurs="0" maxOccurs="1"/>
            <xs:element name="integratedLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="integratedDialogLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="integratedNonDialogLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="momentaryLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="shortTermLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="instantaneousLoudness" type="bpx:LoudnessType" minOccurs="0"
maxOccurs="1"/>
            <xs:element name="loudnessRange" type="bpx:LoudnessType" minOccurs="0" maxOccurs="1"/>
            <xs:element name="truePeak" type="bpx:LoudnessType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="LoudnessType">
        <xs:sequence>
            <xs:element name="value" type="mda:Rational" minOccurs="1" maxOccurs="unbounded"/>
            <xs:element name="units" type="mda:URI"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="DRCType">
        <xs:sequence>
            <xs:element name="DRCProfile" type="bpx:DRCCurve" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="dialogGain" type="mda:Rational" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="DRCCurve">
        <xs:sequence>
            <xs:element name="profileURI" type="mda:URI"/>
            <xs:element name="curve" minOccurs="0">
                <xs:complexType>
                     <xs:attribute name="units" type="mda:URI" use="required"/>
                    <xs:sequence minOccurs="1" maxOccurs="unbounded">
                        <xs:element name="In" type="xs:negativeInteger"/>
                        <xs:element name="Out" type="xs:negativeInteger"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="InteractivityType">
        <xs:sequence>
            <xs:element name="azimuthDelta" type="mda:Rational" minOccurs="0" maxOccurs="1"
default="0"/>
            <xs:element name="elevationDelta" type="mda:Rational" minOccurs="0" maxOccurs="1"
default="0"/>
            <xs:element name="apertureDelta" type="mda:Rational" minOccurs="0" maxOccurs="1"
default="0"/>
            <xs:element name="divergenceDelta" type="mda:Rational" minOccurs="0" maxOccurs="1"
default="0"/>
            <xs:element name="gainDelta" type="mda:Rational" minOccurs="0" maxOccurs="1"
default="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="LockType">
        <xs:sequence>
            <xs:element name="locker" type="mda:URI" minOccurs="0" maxOccurs="1"/>
            <xs:element name="keyID" type="xs:anyURI" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="GroupBroadcastExtension">
        <xs:sequence>
            <xs:element name="extensionURI" type="mda:URI"
fixed="http://mdaif.org/bpx/1.0/extension/group-extension"/>
            <xs:element name="groupKind" type="mda:URI" minOccurs="0" maxOccurs="1"/>
            <xs:element name="groupKindParameters" type="xs:anyType" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
```

```
    </xs:complexType>

</xs:schema>
```

Annex C (informative): Bibliography

- EBU 3341: "Loudness Metering: 'EBU Mode' metering to supplement loudness normalization in accordance with EBU R 128".
- EBU 3342: "Loudness Range: A measure to supplement loudness normalization in accordance with EBU R 128".
- EBU 3343: "Practical Guidelines for Production and Implementation in accordance with EBU R 128".
- EBU 3344: "Practical Guidelines for Distribution of Programmes in accordance with EBU R 128".
- IETF RFC 3629 (January 2003): "UTF-8, a transformation format of ISO 10646".
- MDA Program Specification, SMPTE 25CSS Interoperable Immersive Sound, January 2014.
- ATSC Recommended Practice A/85 (2013): "Techniques for Establishing and Maintaining Audio Loudness for Digital Television".

Annex D (informative): Change History

Date	Version	Information about changes
July 2014	1.0	Complete Draft

History

Document history		
V1.1.1	April 2015	Publication