SMPTE REGISTERED DISCLOSURE DOCUMENT

MDA Bitstream Specification

The attached document is a Registered Disclosure Document prepared by the proponent identified below. It has been examined by the appropriate SMPTE Technology Committee and is believed to contain adequate information to satisfy the objectives defined in the Scope, and to be technically consistent.

This document is NOT a Standard, Recommended Practice or Engineering Guideline, and does NOT imply a finding or representation of the Society.

Every attempt has been made to ensure that the information contained in this document is accurate. Errors in this document should be reported to the proponent identified below, with a copy to eng@smpte.org.

All other inquiries in respect of this document, including inquiries as to intellectual property requirements that may be attached to use of the disclosed technology, should be addressed to the proponent identified below.

Proponent contact information:

Scott Smyers
DTS, Inc.
130 Knowles Drive, Suite B
Los Gatos, CA 95032

Email: scott.smyers@dts.com

Approved
August 23, 2016

Table of Contents

Page

Conventions

All sections are normative, unless otherwise indicated. Pseudo-code and property names use font style `courier new`.

The expressions MAY, NEED NOT, SHALL, SHALL NOT, SHOULD, and SHOULD NOT indicate normative behavior as specified in ISO/EIC Directives, Part 2. Edition 6.0, 2011-04.

VERBAL FORM	SEMANTICS
MAY	It is allowed
NEED NOT	It is not required that
SHALL	Is required that
SHALL NOT	Is required to be not
SHOULD	It is recommended that
SHOULD NOT	It is not recommended that

Introduction (Informative)

The Bitstream consists of an ordered sequence of Packets. As illustrated in Figure 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.

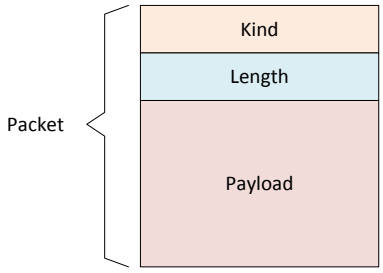

Figure 1 – Packet

Packets are logically grouped into hierarchical structures. Specifically, as depicted in Figure 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 a 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.

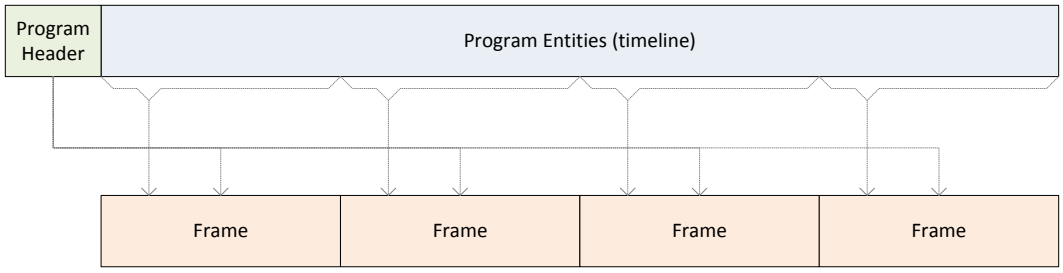

Figure 2 – Mapping Program to Bitstream Frame Structures

As illustrated in Figure 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 of Asset Frame Packets.

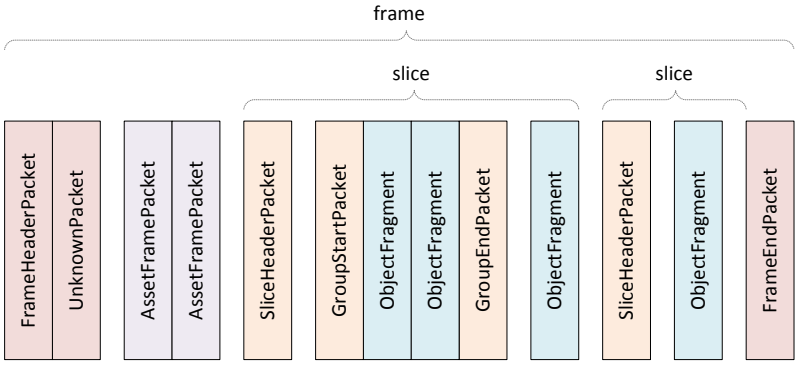

Figure 3 – Frame Structure

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

1 Scope

This specification maps the MDA Program specified in the MDA Program Specification, including audio samples, to a single binary structure – the Bitstream. The Bitstream is partitioned into Frames such that reproduction can start at any Frame boundary, without knowledge of earlier or future Frames.

2 Normative References

MDA Program Specification 1.03

Internet Engineering Task Force (IETF) (January 2005). RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax

ISO/EIC Directives, Part 2. Edition 6.0, 2011-04

Internet Engineering Task Force (IETF) (January 2003). RFC 3629 – UTF-8, A Transformation Format of ISO 10646

3 Structures

The Bitstream syntax is expressed using the operation of a hypothetical parser using the structure specification language described in Section 10.

For extensibility, the Bitstream syntax allows the presence of unknown Packets – captured by fields of type `UnknownPacket`. Implementations MAY ignore these unknown Packets.

3.1 Bitstream

A Bitstream consists of an ordered sequence of Frames, as specified in Table 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}.

Table 1 – Bitstream Structure

```
aligned struct Bitstream {

 var long unsigned int i = 0;

 while(!eof) {

  peek PacketHeader fUnknownHeader;

  // Frames

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

  // Unknown packets
```

```
   else {
     UnknownPacket     fUnknownPacket;
    }
  }
}
```

3.2 Frames

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

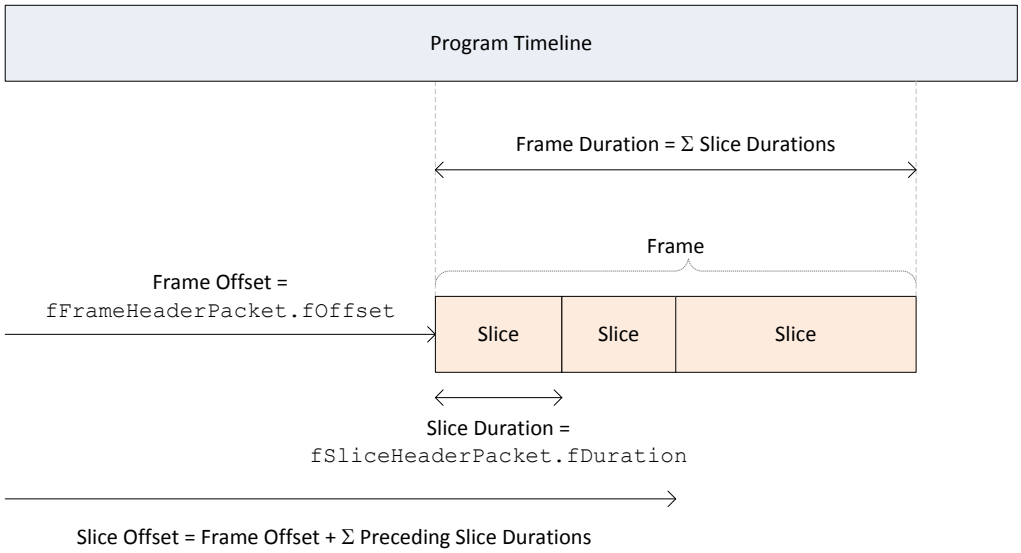

Figure 4 – Frame and Slice Structure Intervals

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 this specification 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 in the range [0,2^{32}].

The total number of Asset Frame Packets in Frame SHALL be in the range [0,2^{32}].

Table 2 – Frame Structure

```
aligned struct Frame {
 // process Frame Header packet
 FrameHeaderPacket fFrameHeaderPacket;
 // process asset frame packets
 var unsigned int i = 0;
 while(!eof) {
  peek PacketHeader  fUnknownHeader;
  if (fUnknownHeader.fKind == AssetFramePacket.fKind) {
   // Asset Packets
   AssetFramePacket fAssetFramePacket[i++];
  } else if (fUnknownHeader.fKind == SliceHeaderPacket.fKind) {
   // End of assets, start of slices
   break;
  } else {
   // Unknown packet
   UnknownPacket fUnknownPacket;
  }
 }
 /* process Slices */
 var unsigned 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 {
    UnknownPacket fUnknownPacket;
   }
  }
}
```

3.3 Slice

The number of Entities in a Slice SHALL be less than or equal to 2^{32}.

The offset and duration of each Entity SHALL be equal to the 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 durations of the Slices preceding it.

Note: The Slice is bitstream-specific structure that does not have a corresponding concept in the MDA Program object model.

Table 3 – Slice Structure

```
aligned struct Slice {
 SliceHeaderPacket fSliceHeaderPacket;

 // process entities

 var unsigned int i = 0;

 while(! eof) {

  peek PacketHeader  fUnknownHeader;


  if (fUnknownHeader.fKind == SliceHeaderPacket.fKind) {

   // SliceHeader, new Slice
```

```
    break;
   }else if (fUnknownHeader.fKind == FrameEndPacket.fKind) {
    // FrameEnd
    break;
   } else if (mIsEntityKind(fUnknownHeader.fKind)) {
    // Entity
    Entity fEntity[i++];
   } else {
    // Unknown packet
    UnknownPacket          fUnknownPacket;
   }
  }
}
```

3.4 Entity

The fields of the `Entity` structure are mapped to concrete derived classes of the `MDA::Entity` abstract class, as summarized in Table 4.

Table 4 – Entity Fields to MDA Object Classes

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

Table 5 defines a function that returns whether a Packet corresponds to an entity defined.

Table 5 – mIsEntityKind() Definition

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

Table 6 – Entity Structure

```
aligned struct Entity {
 peek PacketHeader    fUnknownHeader;

 if (fUnknownHeader.fKind == ObjectFragmentPacket.fKind) {

  // ObjectFragment

  ObjectFragment           fObjectFragment;

 } else if (fUnknownHeader.fKind == LFEFragmentPacket.fKind) {

  // LFEFragment

  LFEFragment              fLFEFragment;

 }else if (fUnknownHeader.fKind == GroupStartPacket.fKind) {

  // Group

  peek GroupStartPacket    fGroupStart;

  if (gGroupStart.fExtensions.fPresent) {
     var uint32 i;

     for(i = 0; i < gGroupStart.fExtensions.fValue.fCount; i++) {
      if (gGroupStart.fExtensions.fValue.fItems[i].fName ==
         kNormalizedGroupExtensionName) {

        NormalizedGroup          fNormalizedGroup;

        break;
      }
     }
  }

  Group                    fGroup;

 } else if (fUnknownHeader.fKind == SwitchStartPacket.fKind) {

  // Switch

  Switch                   fSwitch;

 }
}
```

3.5 LFEFragment

Table 7 – LFEFragment Structure Syntax

```
aligned struct LFEFragment {

 LFEFragmentPacket    fLFEFragmentPacket;

}
```

`LFEFragment` SHALL be mapped to an `MDA::LFEFragment` instance according to Table 8.

Table 8 – Mapping LFEFragment to MDA::LFEFragment

LFEFragment **field**	MDA::LFEFragment **property**
`fLFEFragmentPacket.fId`	See Section 4.12.1
`fLFEFragmentPacket .fExtensions`	See Section 4.12.2
`fLFEFragmentPacket .fAssetURI`	`audioSamples.assetURI`
`fLFEFragmentPacket .fAssetOffset`	`audioSamples.assetOffset`
`fGain = GainFToQ(MDA::LFEFragment.gain)`	`gain = GainQToF(fGain)`

3.6 ObjectFragment

Table 9 – ObjectFragment Structure Syntax

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

An `ObjectFragment` structure SHALL be mapped to a single `MDA::ObjectFragment` instance with `MDA::ObjectFragment.sources.size()` == 1 and according to Table 10.

Table 10 – Mapping ObjectFragment to MDA::ObjectFragment

`ObjectFragment` **field**	`MDA::ObjectFragment` **property**
`fSource.fId`	See Section 4.12.1
`fSource.fExtensions`	See Section 4.12.2
`fSource.fAssetURI`	`audioSamples.assetURI`
`fSource.fAssetOffset`	`audioSamples.assetOffset`
`fCoherent`	`coherent`
`fContentKind`	`contentKind`
`fSource.fGain = GainFToQ(fGain)`	`sources[0].gain = GainQToF(fSource.fGain)`
`fSource.fPosition`	`sources[0].position` (see Section 5.8)
`fSource.fAperture = ApertureFToQ(MDA::LFEFragment.aperture)`	`sources[0].aperture = ApertureQToF(fSource.fAperture)`

fSource.fDivergence = ApertureFToQ(MDA::LFEFragment.aperture)	sources[0].divergence = DivergenceFToQ(fSource.fDivergence)
fChannelExceptions	All ChannelRenderingException instances in sources[0].renderingExceptions
fPositionExceptions	All PositionRenderingException instances in sources[0].renderingExceptions

3.7 Group

Table 11 – Group Entity Structure

```
aligned struct Group {
 GroupStartPacket    fGroupStartPacket;

 // Members

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

  if (fUnknownHeader.fKind == GroupEndPacket.fKind) {

   // End of Group

   GroupEndPacket fGroupEndPacket;
   break;

  }else if (mIsEntityKind(fUnknownHeader.fKind)) {

   // Member

   Entity fMember[i++];
  }else {

   // Ignore Unknown Packet

   UnknownPacket          fUnknownPacket;
  }
 }
}
```

A Group structure shall be mapped to a single MDA::Group instance, with each member of the fMember[] collection corresponding to a member of the MDA::Group.members collection.

3.8 Switch

Table 12 – Switch Entity Syntax

```
aligned struct Switch {
  SwitchStartPacket    fSwitchStartPacket;

  // Members

  var int i = 0;

  while(! eof) {

    peek PacketHeader  fUnknownHeader;

    if (fUnknownHeader.fKind == SwitchEndPacket.fKind) {

      // End of Switch

      SwitchEndPacket fSwitchEndPacket;

      break;

    } else if (mIsEntityKind(fUnknownHeader.fKind)) {

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

      Entity fMember[i++];

    }else {

      // Ignore Unknown packet

      UnknownPacket fUnknownPacket;

    }
  }
}
```

A `Switch` structure shall be mapped to a single `MDA::Switch` instance with:

- The `fMember[0]` member shall correspond to the `MDA::Switch.default` property, and
- each remaining member of the `fMember[]` collection shall correspond to a member of the `MDA::Switch.alternates` collection.

3.9 NormalizedGroup

Table 13 – Group Entity Structure

```
aligned struct NormalizedGroup {
 GroupStartPacket    fGroupStartPacket;
 // Members
 var int i = 0;
 while(! eof) {
  peek PacketHeader  fUnknownHeader;
  // End of Group
  if (fUnknownHeader.fKind == GroupEndPacket.fKind) {
   GroupEndPacket fGroupEndPacket;
   break;
  }else if (fUnknownHeader.fKind == ObjectFragmentPacket) {
   ObjectFragmentPacket fSources[i++];
  }else if (mIsEntityKind(fUnknownHeader.fKind)) {
   // ignore other known packets
  } else {
   UnknownPacket          fUnknownPacket;
  }
 }
}
```

A `NormalizedGroup` structure SHALL be mapped to a single `MDA::ObjectFragment` instance with `MDA::ObjectFragment.sources.size()` equal to `fSources.size()` and according to Table 10.

Table 14 – Mapping ObjectFragment to MDA::ObjectFragment

NormalizedGroup field	MDA::ObjectFragment property
fGroupStartPacket.fId	See Section 4.12.1
fGroupStartPacket.fExtensions	See Section 4.12.1
fSources[0].fAssetURI	audioSamples.assetURI
fSources[0].fAssetOffset	audioSamples.assetOffset
fSources[0].fCoherent	coherent
fSources[0].fContentKind	contentKind
fSources[i].fGain = GainFToQ(MDA::ObjectFragment.sources[i].gain)	sources[i].gain = GainQToF(fSource.fGain)
fSources[i].fPosition	sources[i].position (see Section 5.8)
fSources[i].fAperture = ApertureFToQ(MDA::LFEFragment.sources[i].aperture)	sources[i].aperture = ApertureQToF(fSource.fAperture)
fSources[i].fDivergence = DivergenceFToQ(MDA::LFEFragment.sources[i].divergence)	sources[i].divergence = DivergenceFToQ(fSource.fDivergence)
fSources[i].fChannelExceptions	All MDA::ChannelRenderingException instances in sources[i].renderingExceptions
fSources[i].fPositionExceptions	All MDA::PositionRenderingException instances in sources[i].renderingExceptions

4 Packets

4.1 General

All Label values SHALL be URILabels (as defined in Section 5.4) unless specifically specified otherwise in this specification.

4.2 FrameHeaderPacket

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

This version of the Bitstream Specification does not support the MDA::Program.header.constraintSets property.

Table 15 – Frame Header Packet Structure

```
aligned struct FrameHeaderPacket : PacketHeader {
 fKind = ESFRAMEHEADER LOCALLABEL;

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

 // ConstraintSets not supported for fBitstreamVersion =
        kBitstreamVersion3

 OptionalItem<FixedArray<Extension>>  fExtensions;

 // Frame parameters

 PackedUInt64                   fOffset;
 unsigned int(16)                   fDuration;

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

4.2.1 fBitstreamVersion

The `fBitstreamVersion` field SHALL be equal to `kBitstreamVersion3` for a Bitstream conforming to this version of the MDA Bitstream Specification.

4.2.2 fProgramNamespace

For this version of the Bitstream Specification, the `fProgramNamespace` field SHALL be equal to `<mdacore>` as specified in MDA Program Specification.

4.2.3 fProgramURI

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

4.2.4 fSampleRate

The `fSampleRate` field SHALL be equal to `MDA::Program.header.sampleRate.`

Implementations SHALL support the values `<48000Sampling>` and `<96000Sampling>` as defined in MDA Program Specification.

4.2.5 fExtensions

Each item of `fExtensions` field SHALL correspond to an item of the `MDA::Program.header.extensions` collection.

4.2.6 fOffset

The `fOffset` field SHALL be equal to the offset (in samples) of the first sample of the Frame within the Program timeline.

4.2.7 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.

4.2.8 fCRC

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

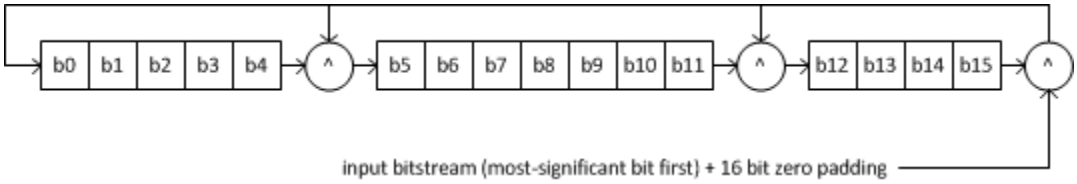

Figure 5 – CRC-16 Algorithm

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

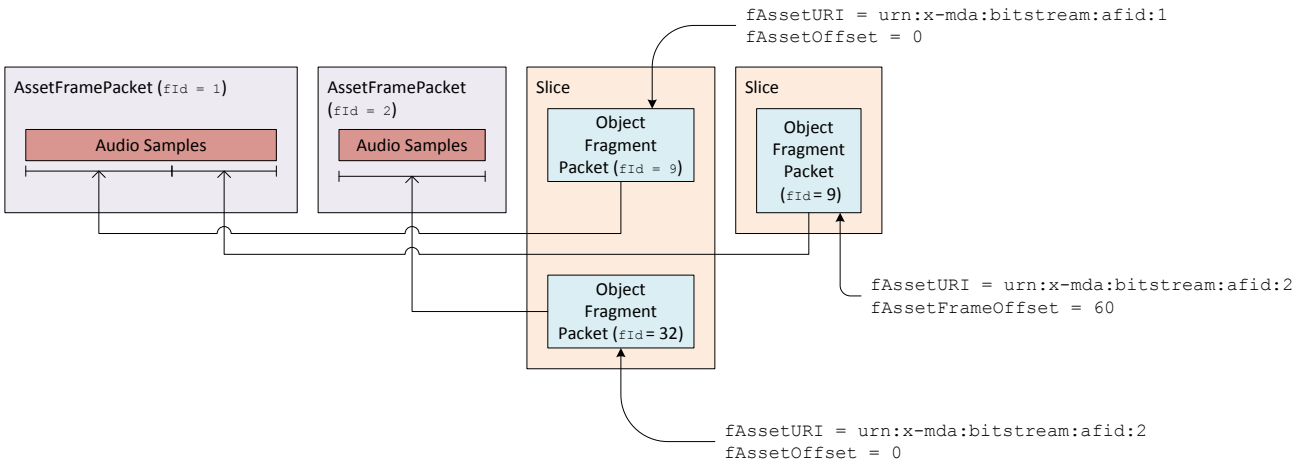

Figure 6 – Relationship between Asset Frame Packets and Object Fragment Packets

4.3 AssetFramePacket

Each `AssetFramePacket` contains a sequence of audio samples associated with the Frame. As illustrated in Figure 6, multiple Entities may reference audio samples within the same `AssetFramePacket`.

Note: The `AssetFramePacket` is bitstream-specific structure that does not have a corresponding concept in the MDA Program object model.

Table 16 – Asset Frame Packet Structure

```
aligned struct AssetFramePacket : PacketHeader {
  fKind = ESASSETFRAME LOCALLABEL;

  PackedUInt32          fId;
  Label                 fAssetEncoding;
  ByteArray                  fAssetBytes;
}
```

4.3.1 fId

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

4.3.2 fAssetEncoding

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

Implementations SHALL support the format `<PCM24BE>` and `<PCM32BE>` listed in Table 17.

Table 17 – Audio Encoding

Constant	URI	Definition
`PCM24BE`	`<mdacore>/labels/essence-encoding/PCM24`	Sequence of 24-bit PCM audio samples, each packed in MSB order.
`PCM32BE`	`<mdacore>/labels/essence-encoding/PCM32`	Sequence of 32-bit PCM audio samples, each packed in MSB order.

4.3.3 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 duration `fFrame[].fDuration` of the containing Frame.

4.4 Frame End Packet

A `FrameEndPacket` signals the end of a Frame.

Table 18 – Frame End Packet Structure

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

4.5 Slice Header Packet

A `SliceHeaderPacket` signals the start of a Slice.

Table 19 – Slice Header Packet Structure

```
aligned struct SliceHeaderPacket : PacketHeader {
  fKind = ESSLICEHEADER LOCALLABEL;

  unsigned int(16)     fDuration;
}
```

4.5.1 fDuration

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

4.6 ObjectFragmentPacket

Table 20 – Object Fragment Packet Structure

```
aligned struct ObjectFragmentPacket : MonoSourceFragmentPacket {
  fKind = ESOBJECTFRAGMENT LOCALLABEL;

  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;

}
```

The `fContentKind` field SHALL be one of the Content Kind values listed in Section 6 of the MDA Program Specification.

4.7 LFEFragmentPacket

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

Table 21 – LFE Fragment Packet Syntax

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

4.8 Group Start Packet

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

Table 22 – GroupStartPacket Syntax

```
aligned struct GroupStartPacket: EntityPacket {
  fKind = ESGROUPSTART LOCALLABEL;
}
```

4.9 Group End Packet

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

Table 23 – GroupEndPacket Syntax

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

4.10 SwitchStartPacket

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

Table 24 – SwitchStartPacket Syntax

```
aligned struct SwitchStartPacket: EntityPacket {

  fKind = ESSWITCHSTART LOCALLABEL;

}
```

4.11 SwitchEndPacket

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

Table 25 – SwitchEndPacket Syntax

```
aligned struct SwitchEndPacket : PacketHeader {

  fKind = ESSWITCHEND LOCALLABEL;

}
```

4.12 EntityPacket

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

Table 26 – EntityPacket Syntax

```
aligned abstract struct EntityPacket : PacketHeader {

  PackedUInt32                    fId;
  OptionalItem<FixedArray<Extension>>  fExtensions;

}
```

4.12.1 fId

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

4.12.2 fExtensions

Each item of `fExtensions` field SHALL correspond to an item of the `MDA::Entity.extensions` collection.

4.13 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 slice to which the `FragmentPacket` belongs.

Table 27 – FragmentPacket Syntax

```
aligned abstract struct FragmentPacket : EntityPacket {

}
```

4.14 MonoSourceFragmentPacket

Table 28 – MonoSourceFragmentPacket Syntax

```
aligned abstract struct MonoSourceFragmentPacket : FragmentPacket {

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

}
```

If `fAssetURI` refers to samples contained in an `AssetFramePacket`, it SHALL be equal to "`urn:x-mda:bitstream:afid:<aid>`" where `<aid>` is the string decimal representation of the `fId` field of the `AssetFramePacket`.

EXAMPLE: `urn:x-mda:bitstream:afid:3` refers to `AssetFramePacket` with `fId = 0x0003;`

Note: Constraints on `fAssetURI` for Frame-external assets are not provided in this specification and depend on the application.

4.15 UnknownPacket

A `UnknownPacket` is a Packet with unspecified payload.

Implementations SHALL accept `UnknownPacket`.

Table 29 – UnknownPacket Syntax

```
aligned struct UnknownPacket : PacketHeader {

 unsigned int(8)      fData[fLength];

}
```

5 Common Data Structures

5.1 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 Section 10.2). 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 this version of the Bitstream Specification (see Table 30).

5.2 ChannelGain

```
struct ChannelGain {

 Label           fChannel;
 unsigned int(8)      fGain;

}
```

5.2.1 fGain

The `fGain` field SHALL be equal to `ChannelGainGToQ(MDA::ChannelGain.gain)`.

The `MDA::ChannelGain.gain` property SHALL be equal to `ChannelGainQToF(fGain)`.

5.3 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;

}
```

5.4 Label

```
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 {

      fKind = kLocalKind;

      unsigned int(4)       fLength;
      unsigned int(8)       fDigits[fLength + 1];

   }

 } else {

   var unsigned int(3)          fKind = kLocalKind;

   unsigned int(7)              fDigits;

 }

}
```

The `Label` structure can carry either a URI as specified in IETF 3986 (a `URILabel`) or a sequence of 8-bit unsigned integers (a `LocalLabel`).

If the `fIsExtended` bit is equal to zero, the remaining bits in the byte provide a 7-bit `LocalLabel`.

It the `fIsExtended` bit is equal to 1, the following 3 bits define the label kind `fKind`:

- If `fKind` is not equal to `kURIKind`, the following 4 bits hold `fLength` and the remaining `fLength` + 1 bytes (`fDigits`) are the `LocalLabel` value.
- If `fKind` is equal to `kURIKind` (=1), the remainder of the Label is a `UTF8String` containing the value of the `URILabel`.

Note: A Label is not guaranteed to be byte aligned, neither is its member `fURI` in case of a `URILabel`.

5.5 PackedLength

```
struct PackedLength {
 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 `PackedLength` 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.

Note: `PackedLength` is not byte aligned.

5.6 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 SHALL 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 this version of the Bitstream Specification.

5.7 FixedArray

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

A `FixedArray` over `T` holds `fCount` items of type `T`.

Note that a `FixedArray` is not guaranteed to be byte aligned, but its members may be.

5.8 Position

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

5.8.1 fRadius

The `fRadius` field and the `MDA::Position.radius` property SHALL be computed according to the following:

```
fRadius = round(2047 · MDA::Position.radius)
MDA::Position.radius = fRadius / 2047
```

The value `fRadius = 4095` is reserved and SHALL NOT be present.

5.8.2 fAzimuth

The `fAzimuth` field and the `MDA::Position.theta` property SHALL be computed according to the following:

```
fAzimuth = round(2048 · MDA::Position.theta / π + 2048)
MDA::Position.theta = (fAzimuth - 2048) · π / 2048
```

5.8.3 fElevation

The `fElevation` field and the `MDA::Position.phi` property SHALL be computed according to the following:

```
fElevation = round(2046 · MDA::Position.phi / π + 1023)
MDA::Position.phi = (fElevation - 1023) · π / 2046
```

The value `fElevation = 2047` is reserved and SHALL NOT be present.

5.9 ByteArray

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

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

5.10 PackedUInt64

```
struct PackedUInt64 {
  unsigned int(3)                       fLength = 7;
  unsigned int((fLength + 1) << 3)      fValue;
}
```

Note 1: For this version of the specification the `fLength` field is always equal to 7, corresponding to 8 bytes in Big Endian order to encode `fValue`.

Note 2: `PackedUInt64` is not byte aligned.

5.11 PackedUInt32

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

Note 1: For this version of the specification the `fLength` field is always equal to 3, corresponding to 4 bytes in Big Endian order to encode `fValue`.

Note 2: `PackeUInt32` is not byte aligned.

5.12 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.

5.13 UTF8String

```
typedef ByteArray UTF8String;
```

The `UTF8String` data type holds UTF8 encoded character strings as specified in IETF RFC 3629.

Note: A `UTF8String` is not guaranteed to be byte aligned within an MDA bitstream.

6 Common Functions

6.1 RadiusQToF

```
RadiusQToF(pRadius) = pRadius / 2047
```

6.2 RadiusFToQ

```
RadiusFToQ(pRadius) = round( pRadius * 2047 )
```

6.3 PhiQToF

```
PhiQToF(pPhi) = (pPhi - 1023) * mdaKPI / 2046)
```

6.4 PhiFToQ

```
PhiFToQ(pPhi) = round( pPhi * 2046 / mdaKPI + 1023 )
```

6.5 ThetaQToF

```
ThetaQToF(pTheta) = ( pTheta – 2048 ) * mdaKPI / 2048
```

6.6 ThetaFToQ

```
ThetaFToQ(pTheta) = round( pTheta * 2048 / mdaKPI + 2048 )
```

6.7 ApertureQToF

```
ApertureQToF(pAperture) = mdaKPI * pAperture / 255
```

6.8 ApertureFToQ

```
ApertureFToQ(pAperture) = round(pAperture / mdaKPI * 255)
```

6.9 DivergenceQToF

```
DivergenceQToF(pDivergence) = mdaKPI * pDivergence / 255
```

6.10 DivergenceFToQ

```
DivergenceFToQ(pDivergence) = round(pDivergence / mdaKPI * 255)
```

6.11 GainQToF

```
GainQToF(pGain) = pGain == 0 ? -∞ : (pGain - 411 ) / 4
```

6.12 GainFToQ

```
GainFToQ(pGain) = pGain == -∞ ? 0 : 4 • pGain + 411
```

6.13 ChannelGainQToF

```
ChannelGainQToF(pGain) = -pGain / 4
```

6.14 ChannelGainFToQ

```
ChannelGainFToQ(pGain) = round(-pGain * 4)
```

7 Constants

7.1 Packet Kind Labels

Table 30 – Packet Kinds

```
const Label ESFRAMEHEADER LOCALLABEL =
          {1,0x00,0x01,{0x5A,0xA5}};
const Label ESFRAMEEND LOCALLABEL =         {0,0x01};
const Label ESSLICEHEADER LOCALLABEL =           {0,0x02};
const Label ESOBJECTFRAGMENT LOCALLABEL =   {0,0x03};
const Label ESASSETFRAME LOCALLABEL =       {0,0x04};
const Label ESGROUPSTART LOCALLABEL =       {0,0x05};
const Label ESGROUPEND LOCALLABEL =         {0,0x06};
const Label ESLFEFRAGMENT LOCALLABEL =           {0,0x07};
const Label ESSWITCHSTART LOCALLABEL =           {0,0x08};
const Label ESSWITCHEND LOCALLABEL =        {0,0x09};
```

7.2 Bitstream Version

Table 31 – Bitstream Version

```
const int kBitstreamVersion3 = 0x03;
```

7.3 Normalized Group Label

Table 32 – Bitstream Version

```
const Label kNormalizedGroupExtensionName = <mdacore>/1.0/extension/norm-
         group;
```

8 Extensibility

Bitstream extensibility can be achieved through two mechanisms.

- First, additional Packets can be introduced as long as their Label values do not conflict with already-defined Packets.
- Second, although not encouraged, additional information can be added at the end of each Packet. This information will be ignored by legacy decoders, which will use the Length field to locate the next Packet.

9 Authoring Guidelines

The following collects Bitstream authoring guidelines.

- Applications will typically impose a constant Frame duration. This will in turn require that Entity instances be split across Bitstream Frames.

10 Structure Specification Language

The structure specification language used in this specification adopts a C-like syntax and is not unlike that used in MPEG standards.

All standard integer mathematical operators are supported.

10.1 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`.

10.2 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.

10.3 Basic Type

The following signed and unsigned integer types are defined.

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

10.4 Type Aliasing

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

```
typedef NameOfType NameOfAliasedType;
```

10.5 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.

10.6 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;
```

10.7 Variables

A Variable is declared as follows.

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

A Variable is never read from the stream.

10.8 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.