INTERNATIONAL STANDARD

ISO/IEC 14496-12

Third edition
2008-10-15

AMENDMENT 1
2009-11-15

Information technology — Coding of audio-visual objects —

Part 12: ISO base media file format

AMENDMENT 1: General improvements including hint tracks, metadata support and sample groups

Technologies de l'information — Codage des objets audiovisuels —

Partie 12: Format ISO de base pour les fichiers médias

AMENDEMENT 1: Améliorations générales comprenant «hint tracks», support de métadonnées et groupes d'échantillons

Reference number
ISO/IEC 14496-12:2008/Amd.1:2009(E)

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

Published in Switzerland

Foreword

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

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

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

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

Amendment 1 to ISO/IEC 14496-12:2008 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This Amendment provides a number of new hint track formats and improvements in other areas, including metadata support.

Information technology — Coding of audio-visual objects —

Part 12:
ISO base media file format

AMENDMENT 1: General improvements including hint tracks, metadata support and sample groups

Reformat the table of contents, including the new subclauses and revised page numbers.

Add the following normative references to Clause 2:

ISO/IEC 23002-3, *Information technology — MPEG video technologies — Part 3: Representation of auxiliary video streams and supplemental information*

IETF RFC 5052, *Forward Error Correction (FEC) Building Block*, WATSON, M. et al., August 2007

In 6.2.3, Table 1, add the following line after 'tref':

		trgr					8.3.4	*track grouping indication*

In 6.2.3, Table 1, add the following line after 'ctts':

					cslg		8.6.1.4	*composition to decode timeline mapping*

In 6.2.3, add the following after 'paen':

			fire				8.13.7	*file reservoir*

In 6.2.3, Table 1, add the following lines after 'tsel':

	idat						8.11.11	*item data*
	iref						8.11.12	*item reference*

After 6.2.3, add the following new subclause:

6.2.4 URIs as type indicators

When URIs are used as a type indicator (e.g. in a sample entry or for un-timed meta-data), the URI must be absolute, not relative and the format and meaning of the data must be defined by the URI in question. This identification may be hierarchical, in that an initial sub-string of the URI might identify the overall nature or family of the data (e.g. urn:oid: identifies that the metadata is labelled by an ISO-standard object identifier).

The URI should be, but is not required to be, de-referencable. It may be string compared by readers with the set of URI types it knows and recognizes. URIs provide a large non-colliding non-registered space for type identifiers.

If the URI contains a domain name (e.g. it is a URL), then it should also contain a month-date in the form mmyyyy. That date must be near the time of the definition of the extension, and it must be true that the URI was defined in a way authorized by the owner of the domain name at that date. (This avoids problems when domain names change ownership).

Replace Clause 7 with the following:

7 Streaming Support

7.1 Handling of Streaming Protocols

The file format supports streaming of media data over a network as well as local playback. The process of sending protocol data units is time-based, just like the display of time-based data, and is therefore suitably described by a time-based format. A file or 'movie' that supports streaming includes information about the data units to stream. This information is included in additional tracks of the file called "hint" tracks. Hint tracks may also be used to record a stream; these are called Reception Hint Tracks, to differentiate them from plain (or server, or transmission) hint tracks.

Transmission or server hint tracks contain instructions to assist a streaming server in the formation of packets for transmission. These instructions may contain immediate data for the server to send (e.g. header information) or reference segments of the media data. These instructions are encoded in the file in the same way that editing or presentation information is encoded in a file for local playback. Instead of editing or presentation information, information is provided which allows a server to packetize the media data in a manner suitable for streaming using a specific network transport.

The same media data is used in a file that contains hints, whether it is for local playback, or streaming over a number of different protocols. Separate 'hint' tracks for different protocols may be included within the same file and the media will play over all such protocols without making any additional copies of the media itself. In addition, existing media can be easily made streamable by the addition of appropriate hint tracks for specific protocols. The media data itself need not be recast or reformatted in any way.

This approach to streaming and recording is more space efficient than an approach that requires that the media information be partitioned into the actual data units that will be transmitted for a given transport and media format. Under such an approach, local playback requires either re-assembling the media from the packets, or having two copies of the media — one for local playback and one for streaming. Similarly, streaming such media over multiple protocols using this approach requires multiple copies of the media data for each transport. This is inefficient with space, unless the media data has been heavily transformed for streaming (e.g. by the application of error-correcting coding techniques, or by encryption).

Reception hint tracks may be used when one or more packet streams of data are recorded. Reception hint tracks indicate the order, reception timing, and contents of the received packets among other things.

NOTE 1: Players may reproduce the packet stream that was received based on the reception hint tracks and process the reproduced packet stream as if it was newly received.

7.2 Protocol 'hint' tracks

Support for streaming is based upon the following three design parameters:

- The media data is represented as a set of network-independent standard tracks, which may be played, edited, and so on, as normal;

- There is a common declaration and base structure for hint tracks; this common format is protocol independent, but contains the declarations of which protocol(s) are described in the hint track(s);
- There is a specific design of the hint tracks for each protocol that may be transmitted; all these designs use the same basic structure. For example, there may be designs for RTP (for the Internet) and MPEG-2 transport (for broadcast), or for new standard or vendor-specific protocols.

The resulting streams, sent by the servers under the direction of the server hint tracks or reconstructed from the reception hint tracks, need contain no trace of file-specific information. This design does not require that the file structures or declaration style, be used either in the data on the wire or in the decoding station. For example, a file using ITU-T H.261 video and DVI audio, streamed under RTP, results in a packet stream that is fully compliant with the IETF specifications for packing those codings into RTP.

7.3 Hint Track Format

Hint tracks are used to describe elementary stream data in the file. Each protocol or each family of related protocols has its own hint track format. A server hint track format and a reception hint track format for the same protocol are distinguishable from the associated four-character code of the sample description entry. In other words, a different four-character code is used for a server hint track and a reception hint track of the same protocol. The syntax of the server hint track format and the reception hint track format for the same protocol should be the same or compatible so that a reception hint track can be used for re-sending of the stream provided that the potential degradations of the received streams are handled appropriately. Most protocols will need only one sample description format for each track.

Servers find their hint tracks by first finding all hint tracks, and then looking within that set for server hint tracks using their protocol (sample description format). If there are choices at this point, then the server chooses on the basis of preferred protocol or by comparing features in the hint track header or other protocol-specific information in the sample descriptions. Particularly if the absence of server hint tracks, servers may also use reception hint tracks of their protocol. However, servers should handle potential degradations of the received stream described by the used reception hint track appropriately.

Tracks having the track_in_movie flag set are candidates for playback, regardless of whether they are media tracks or reception hint tracks.

Hint tracks construct streams by pulling data out of other tracks by reference. These other tracks may be hint tracks or elementary stream tracks. The exact form of these pointers is defined by the sample format for the protocol, but in general they consist of four pieces of information: a track reference index, a sample number, an offset, and a length. Some of these may be implicit for a particular protocol. These 'pointers' always point to the actual source of the data. If a hint track is built 'on top' of another hint track, then the second hint track must have direct references to the media track(s) used by the first where data from those media tracks is placed in the stream.

All hint tracks use a common set of declarations and structures.

- Hint tracks are linked to the elementary stream tracks they carry, by track references of type `'hint'`
- They use a handler-type of `'hint'` in the Handler Reference Box
- They use a Hint Media Header Box
- They use a hint sample entry in the sample description, with a name and format unique to the protocol they represent.

Server hint tracks are usually marked as disabled for local playback, with their track header `track_in_movie` and `track_in_preview` flags set to 0.

Hint tracks may be created by an authoring tool, or may be added to an existing presentation by a hinting tool. Such a tool serves as a 'bridge' between the media and the protocol, since it intimately understands both. This

permits authoring tools to understand the media format, but not protocols, and for servers to understand protocols (and their hint tracks) but not the details of media data.

Hint tracks do not use separate composition times; the `'ctts'` table is not present in hint tracks. The process of hinting computes transmission times correctly as the decoding time.

NOTE 1: Servers using reception hint tracks as hints for sending of the received streams should handle the potential degradations of the received streams, such as transmission delay jitter and packet losses, gracefully and ensure that the constraints of the protocols and contained data formats are obeyed regardless of the potential degradations of the received streams.

NOTE 2: Conversion of received streams to media tracks allows existing players compliant with earlier versions of the ISO base media file format to process recorded files as long as the media formats are supported. However, most media coding standards only specify the decoding of error-free streams, and consequently it should be ensured that the content in media tracks can be correctly decoded. Players may utilize reception hint tracks for handling of degradations caused by the transmission, i.e., content that may not be correctly decoded is located only within reception hint tracks. The need for having a duplicate of the correct media samples in both a media track and a reception hint track can be avoided by including data from the media track by reference into the reception hint track.

In 8.3.2.1, change:

Hint tracks should have the track header flags set to 0, so that they are ignored for local playback and preview.

to:

Server hint tracks should have the `track_in_movie` and `track_in_preview` track header flags set to 0, so that they are ignored for local playback and preview.

In 8.3.3.3 (Track Reference Box, Semantics), replace the definition of reference_type as follows:

The `reference_type` shall be set to one of the following values, or a value registered or from a derived specification or registration:

- 'hint' the referenced track(s) contain the original media for this hint track
- 'cdsc' this track describes the referenced track.
- 'hind' this track depends on the referenced hint track, i.e., it should only be used if the referenced hint track is used.
- 'vdep' this track contains auxiliary depth video information for the referenced video track
- 'vplx' this track contains auxiliary parallax video information for the referenced video track

After 8.3.3, add the following new subclause:

8.3.4 Track Group Box

8.3.4.1 Definition

Box Type: 'trgr'
Container: Track Box (`'trak'`)
Mandatory: No
Quantity: Zero or one

This box enables indication of groups of tracks, where each group shares a particular characteristic or the tracks within a group have a particular relationship. The box contains zero or more boxes, and the particular

characteristic or the relationship is indicated by the box type of the contained boxes. The contained boxes include an identifier, which can be used to conclude the tracks belonging to the same track group. The tracks that contain the same type of a contained box within the Track Group Box and have the same identifier value within these contained boxes belong to the same track group.

Track groups shall not be used to indicate dependency relationships between tracks. Instead, the Track Reference Box is used for such purposes.

8.3.4.2 Syntax

```
aligned(8) class TrackGroupBox('trgr') {
}

aligned(8) class TrackGroupTypeBox(unsigned int(32) track_group_type) extends
FullBox(track_group_type, version = 0, flags = 0)
{
   unsigned int(32) track_group_id;
   // the remaining data may be specified for a particular track_group_type
}
```

8.3.4.3 Semantics

`track_group_type` indicates the grouping type and shall be set to one of the following values, or a value registered, or a value from a derived specification or registration:

- `'msrc'` indicates that this track belongs to a multi-source presentation. The tracks that have the same value of `track_group_id` within a Group Type Box of `track_group_type` `'msrc'` are mapped as being originated from the same source. For example, a recording of a video telephony call may have both audio and video for both participants, and the value of `track_group_id` associated with the audio track and the video track of one participant differs from value of `track_group_id` associated with the tracks of the other participant.

The pair of `track_group_id` and `track_group_type` identifies a track group within the file. The tracks that contain a particular track group type box having the same value of `track_group_id` belong to the same track group.

In 8.4.3.1, (Handler Reference Box, definition), before the NOTEs, add the following paragraph:

An auxiliary video track is coded the same as a video track, but uses this different handler type, and is not intended to be visually displayed (e.g. it contains depth information, or other monochrome or color two-dimensional information). Auxiliary video tracks are usually linked to a video track by an appropriate track reference.

In 8.4.3.3, after the line for 'meta' add the following line:

`'auxv'` Auxiliary Video track

At the end of 8.5.2.1 (Sample Description, Definition), add the following:

The URIMetaSampleEntry entry contains, in a box, the URI defining the form of the metadata, and optional initialization data. The format of both the samples and of the initialization data is defined by all or part of the URI form.

An optional bitrate box may be used in the URIMetaSampleEntry entry, as usual.

It may be the case that the URI identifies a format of metadata that allows there to be more than one 'stated fact' within each sample. However, all metadata samples in this format are effectively 'I frames', defining the entire set of metadata for the time interval they cover. This means that the complete set of metadata at any

instant, for a given track, is contained in (a) the time-aligned samples of the track(s) (if any) describing that track, plus (b) the track metadata (if any), the movie metadata (if any) and the file metadata (if any).

If incrementally-changed metadata is needed, the MPEG-7 framework provides that capability.

Information on URI forms for some metadata systems can be found in Annex G.

In 8.5.2.2, add before "// Visual Sequences":

```
aligned(8) class URIBox
      extends FullBox('uri ', version = 0, 0) {
   string   theURI;
}

aligned(8) class URIInitBox
      extends FullBox('uriI', version = 0, 0) {
   unsigned int(8) uri_initialization_data[];
}

class URIMetaSampleEntry() extends MetaDataSampleEntry ('urim'){
   URIbox         the_label;
   URIInitBox     init;     // optional
   MPEG4BitRateBox ();      // optional
}
```

At the end of 8.5.2.3, add the following:

`theURI` is a URI formatted according to the rules in 6.2.4;
`uri_initialization_data` is opaque data whose form is defined in the documentation of the URI form.

In 8.6.1.3.1 (Composition Time to Sample, Definition), replace the body text with the following:

This box provides the offset between decoding time and composition time. In version 0 of this box the decoding time must be less than the composition time, and the offsets are expressed as unsigned numbers such that CT(n) = DT(n) + CTTS(n) where CTTS(n) is the (uncompressed) table entry for sample n. In version 1 of this box, the composition timeline and the decoding timeline are still derived from each other, but the offsets are signed. It is recommended that for the computed composition timestamps, there is exactly one with the value 0 (zero).

For either version of the box, each sample must have a unique composition timestamp value, that is, the timestamp for two samples shall never be the same.

It may be true that there is no frame to compose at time 0; the handling of this is unspecified (systems might display the first frame for longer, or a suitable fill colour).

When version 1 of this box is used, the CompositionToDecodeBox may also be present in the sample table to relate the composition and decoding timelines. When backwards-compatibility or compatibility with an unknown set of readers is desired,, version 0 of this box should be used when possible. In either version of this box, but particularly under version 0, if it is desired that the media start at track time 0, and the first media sample does not have a composition time of 0, an edit list may be used to 'shift' the media to time 0.

The composition time to sample table is optional and must only be present if DT and CT differ for any samples.

Hint tracks do not use this box.

Move the table and its header 'For example in Table 2' from 8.6.1.3.2 to the end of 8.6.1.3.1

Replace 8.6.1.3.2 with the following:

8.6.1.3.2 Syntax

```
aligned(8) class CompositionOffsetBox
   extends FullBox('ctts', version = 0, 0) {
   unsigned int(32)  entry_count;
      int i;
   if (version==0) {
      for (i=0; i < entry_count; i++) {
         unsigned int(32)  sample_count;
         unsigned int(32)  sample_offset;
      }
   }
   else if (version == 1) {
      for (i=0; i < entry_count; i++) {
         unsigned int(32)  sample_count;
         signed   int(32)  sample_offset;
      }
   }
}
```

In 8.6.1.3.3, replace the following line with this revised definition:

`sample_offset` is an integer that gives the offset between CT and DT, such that CT(n) = DT(n) + CTTS(n).

After 8.6.1.3, add the following:

8.6.1.4 Composition to Decode Box

8.6.1.4.1 Definition

Box Type: `'cslg'`
Container: Sample Table Box (`'stbl'`)
Mandatory: No
Quantity: Zero or one

When signed composition offsets are used, this box may be used to relate the composition and decoding timelines, and deal with some of the ambiguities that signed composition offsets introduce.

Note that all these fields apply to the entire media (not just that selected by any edits). It is recommended that any edits, explicit or implied, not select any portion of the composition timeline that does not map to a sample. For example, if the smallest composition time is 1000, then the default edit from 0 to the media duration leaves the period from 0 to 1000 associated with no media sample. Player behaviour, and what is composed in this interval, is undefined under these circumstances. It is recommended that the smallest computed CTS be zero, or match the beginning of the first edit.

The composition duration of the last sample in a track might be (often is) ambiguous or unclear; the field for composition end time can be used to clarify this ambiguity and, with the composition start time, establish a clear composition duration for the track.

8.6.1.4.2 Syntax

```
class CompositionToDecodeBox extends FullBox('cslg', version=0, 0) {
   signed int(32) compositionToDTSShift;
   signed int(32) leastDecodeToDisplayDelta;
   signed int(32) greatestDecodeToDisplayDelta;
   signed int(32) compositionStartTime;
   signed int(32) compositionEndTime;
}
```

8.6.1.4.3 Semantics

`compositionToDTSShift`: if this value is added to the composition times (as calculated by the CTS offsets from the DTS), then for all samples, their CTS is guaranteed to be greater than or equal to their DTS, and the buffer model implied by the indicated profile/level will be honoured; if `leastDecodeToDisplayDelta` is positive or zero, this field can be 0; otherwise it should be at least `(- leastDecodeToDisplayDelta)`

`leastDecodeToDisplayDelta`: the smallest composition offset in the CompositionTimeToSample box in this track

`greatestDecodeToDisplayDelta`: the largest composition offset in the CompositionTimeToSample box in this track

`compositionStartTime`: the smallest computed composition time (CTS) for any sample in the media of this track

`compositionEndTime`: the composition time plus the composition duration, of the sample with the largest computed composition time (CTS) in the media of this track

In 8.6.4.1 (Independent and Disposable Samples, Definition), add the following paragraph before the penultimate paragraph (starting "The size of the table"):

A leading sample (usually a picture in video) is defined relative to a reference sample, which is the immediately prior sample that is marked as "sample_depends_on" having no dependency (an I picture). A leading sample has both a composition time before the reference sample, and possibly also a decoding dependency on a sample before the reference sample. Therefore if, for example, playback and decoding were to start at the reference sample, those samples marked as leading would not be needed and might not be decodable. A leading sample itself must therefore not be marked as having no dependency.

Replace 8.6.4.2 with the following:

8.6.4.2 Syntax

```
aligned(8) class SampleDependencyTypeBox
   extends FullBox('sdtp', version = 0, 0) {
   for (i=0; i < sample_count; i++){
      unsigned int(2) is_leading;
      unsigned int(2) sample_depends_on;
      unsigned int(2) sample_is_depended_on;
      unsigned int(2) sample_has_redundancy;
   }
}
```

Add the following to the beginning of 8.6.4.3:

`is_leading` takes one of the following four values:

0: the leading nature of this sample is unknown;

1: this sample is a leading sample that has a dependency before the referenced I-picture (and is therefore not decodable);

2: this sample is not a leading sample;

3: this sample is a leading sample that has no dependency before the referenced I-picture (and is therefore decodable);

In 8.8.3.1 (Track Extends, definition), replace the code and the following paragraph, with the following:

```
        bit(4)    reserved=0;
        unsigned int(2) is_leading;
        unsigned int(2) sample_depends_on;
        unsigned int(2) sample_is_depended_on;
        unsigned int(2) sample_has_redundancy;
        bit(3)    sample_padding_value;
        bit(1)    sample_is_difference_sample;
              // i.e. when 1 signals a non-key or non-sync sample
        unsigned int(16)  sample_degradation_priority;
```

The `is_leading`, `sample_depends_on`, `sample_is_depended_on` and `sample_has_redundancy` values are defined as documented in the Independent and Disposable Samples Box.

In 8.9.1 (Sample Group Structures, Introduction), add the following before the final paragraph (that starts "One example"):

A grouping of a particular grouping type may use a parameter in the sample to group mapping; if so, the meaning of the parameter must be documented with the group. An example of this might be documented the sync points in a multiplex of several video streams; the group definition might be 'Is an I frame', and the group parameter might be the identifier of each stream. Since the sample to group box occurs once for each stream, it is now both compact, and informs the reader about each stream separately.

Add the following to the end of 8.9.2.1 (Sample to Group, Definition):

Version 1 of this box should only be used if a grouping type parameter is needed.

Replace 8.9.2.2 as follows:

8.9.2.2 Syntax

```
aligned(8) class SampleToGroupBox
   extends FullBox('sbgp', version, 0)
{
   unsigned int(32)  grouping_type;
   if (version == 1) {
      unsigned int(32) grouping_type_parameter;
   }
   unsigned int(32)  entry_count;
   for (i=1; i <= entry_count; i++)
   {
      unsigned int(32)  sample_count;
      unsigned int(32)  group_description_index;
   }
}
```

Replace the first two paragraphs (preceding "entry_count") of 8.9.2.3 with the following:

`version` is an integer that specifies the version of this box, either 0 or 1.

`grouping_type` is an integer that identifies the type (i.e. criterion used to form the sample groups) of the sample grouping and links it to its sample group description table with the same value for grouping type. At most one occurrence of this box with the same value for `grouping_type` (and, if used, `grouping_type_parameter`) shall exist for a track.

`grouping_type_parameter` is an indication of the sub-type of the grouping

Replace 8.11.1.2 and 8.11.1.3 (Meta Box, Syntax and Semantics) with the following:

8.11.1.2 Syntax

```
aligned(8) class MetaBox (handler_type)
   extends FullBox('meta', version = 0, 0) {
   HandlerBox(handler_type)   theHandler;
   PrimaryItemBox       primary_resource;       // optional
   DataInformationBox   file_locations;         // optional
   ItemLocationBox      item_locations;         // optional
   ItemProtectionBox    protections;            // optional
   ItemInfoBox          item_infos;             // optional
   IPMPControlBox       IPMP_control;           // optional
   ItemReferenceBox     item_refs;              // optional
   ItemDataBox          item_data;              // optional
   Box   other_boxes[];                         // optional
}
```

8.11.1.3 Semantics

The structure or format of the metadata is declared by the handler. In the case that the primary data is identified by a primary item, and that primary item has an item information entry with an `item_type`, the handler type may be the same as the `item_type`.

Replace 8.11.3 with the following:

8.11.3 The Item Location Box

8.11.3.1 Definition

Box Type: `'iloc'`
Container: Meta box (`'meta'`)
Mandatory: No
Quantity: Zero or one

The item location box provides a directory of resources in this or other files, by locating their containing file, their offset within that file, and their length. Placing this in binary format enables common handling of this data, even by systems which do not understand the particular metadata system (handler) used. For example, a system might integrate all the externally referenced metadata resources into one file, re-adjusting file offsets and file references accordingly.

The box starts with three or four values, specifying the size in bytes of the `offset` field, `length` field, `base_offset` field, and in version 1 of this box, the `extent_index` fields, respectively. These values must be from the set {0, 4, 8}.

The `construction_method` field indicates the 'construction method' for the item:

i) file_offset: by the usual absolute file offsets into the file at data_reference_index; (construction_method == 0)

ii) idat_offset: by box offsets into the idat box in the same meta box; neither the data_reference_index nor extent_index fields are used; (construction_method == 1)

iii) item_offset: by item offset into the items indicated by a new extent_index field, which is only used (currently) by this construction method. (construction_method == 2).

The extent_index is only used for the method item_offset; it indicates the 1-based index of the item reference with referenceType 'iloc' linked from this item. If index_size is 0, then the value 1 is implied; the value 0 is reserved.

Items may be stored fragmented into extents, e.g. to enable interleaving. An extent is a contiguous subset of the bytes of the resource; the resource is formed by concatenating the extents. If only one extent is used (`extent_count` = 1) then either or both of the offset and length may be implied:

- If the offset is not identified (the field has a length of zero), then the beginning of the source (offset 0 into the file, idat box, or other item) is implied.

- If the length is not specified, or specified as zero, then the entire length of the source is implied. References into the same file as this metadata, or items divided into more than one extent, should have an explicit offset and length, or use a MIME type requiring a different interpretation of the file, to avoid infinite recursion.

The size of the item is the sum of the extent lengths.

NOTE Extents may be interleaved with the chunks defined by the sample tables of tracks.

The data-reference index may take the value 0, indicating a reference into the same file as this metadata, or an index into the data-reference table.

Some referenced data may itself use offset/length techniques to address resources within it (e.g. an MP4 file might be 'included' in this way). Normally such offsets are relative to the beginning of the containing file. The field 'base offset' provides an additional offset for offset calculations within that contained data. For example, if an MP4 file is included within a file formatted to this specification, then normally data-offsets within that MP4 section are relative to the beginning of file; the base offset adds to those offsets.

If an item is constructed from other items, and those source items are protected, the offset and length information apply to the source items after they have been de-protected. That is, the target item data is formed from unprotected source data.

For maximum compatibility, version 0 of this box should be used in preference to version 1 with construction_method==0, when possible.

8.11.3.2 Syntax

```
aligned(8) class ItemLocationBox extends FullBox('iloc', version, 0) {
   unsigned int(4)   offset_size;
   unsigned int(4)   length_size;
   unsigned int(4)   base_offset_size;
   if (version == 1)
      unsigned int(4)   index_size;
   else
      unsigned int(4)   reserved;
   unsigned int(16)  item_count;
   for (i=0; i<item_count; i++) {
      unsigned int(16)  item_ID;
      if (version == 1) {
         unsigned int(12)  reserved = 0;
         unsigned int(4)   construction_method;
      }
      unsigned int(16)  data_reference_index;
      unsigned int(base_offset_size*8)  base_offset;
      unsigned int(16)     extent_count;
      for (j=0; j<extent_count; j++) {
         if ((version == 1) && (index_size > 0)) {
            unsigned int(index_size*8)  extent_index;
         }
         unsigned int(offset_size*8) extent_offset;
         unsigned int(length_size*8) extent_length;
      }
   }
}
```

8.11.3.3 Semantics

`offset_size` is taken from the set {0, 4, 8} and indicates the length in bytes of the `offset` field.
`length_size` is taken from the set {0, 4, 8} and indicates the length in bytes of the `length` field.
`base_offset_size` is taken from the set {0, 4, 8} and indicates the length in bytes of the `base_offset` field.
`index_size` is taken from the set {0, 4, 8} and indicates the length in bytes of the `extent_index` field.
`item_count` counts the number of resources in the following array.
`item_ID` is an arbitrary integer 'name' for this resource which can be used to refer to it (e.g. in a URL).
`construction_method` is taken from the set 0 (file), 1 (idat) or 2 (item)
`data-reference-index` is either zero ('this file') or a 1-based index into the data references in the data information box.
`base_offset` provides a base value for offset calculations within the referenced data. If `base_offset_size` is 0, `base_offset` takes the value 0, i.e. it is unused.
`extent_count` provides the count of the number of extents into which the resource is fragmented; it must have the value 1 or greater
`extent_index` provides an index as defined for the construction method
`extent_offset` provides the absolute offset in bytes from the beginning of the containing file, of this item. If `offset_size` is 0, `offset` takes the value 0
`extent_length` provides the absolute length in bytes of this metadata item. If `length_size` is 0, `length` takes the value 0. If the value is 0, then length of the item is the length of the entire referenced file.

Replace 8.11.6 with the following:

8.11.6 Item Information Box

8.11.6.1 Definition

Box Type: `'iinf'`
Container: Meta Box (`'meta'`)
Mandatory: No
Quantity: Zero or one

The Item information box provides extra information about selected items, including symbolic ('file') names. It may optionally occur, but if it does, it must be interpreted, as item protection or content encoding may have changed the format of the data in the item. If both content encoding and protection are indicated for an item, a reader should first un-protect the item, and then decode the item's content encoding. If more control is needed, an IPMP sequence code may be used.

This box contains an array of entries, and each entry is formatted as a box. This array is sorted by increasing `item_ID` in the entry records.

Three versions of the item info entry are defined. Version 1 includes additional information to version 0 as specified by an extension type. For instance, it shall be used with extension type `'fdel'` for items that are referenced by the file partition box (`'fpar'`), which is defined for source file partitionings and applies to file delivery transmissions. Version 2 provides an alternative structure in which metadata item types are indicated by a 32-bit (typically 4-character) registered or defined code; two of these codes are defined to indicate a MIME type or metadata typed by a URI.

If no extension is desired, the box may terminate without the `extension_type` field and the extension; if, in addition, `content_encoding` is not desired, that field also may be absent and the box terminate before it. If an extension is desired without an explicit `content_encoding`, a single null byte, signifying the empty string, must be supplied for the `content_encoding`, before the indication of `extension_type`.

If file delivery item information is needed and a version 2 ItemInfoEntry is used, then the file delivery information is stored (a) as a separate item of type 'fdel') (b) linked by an item reference from the item, to the file delivery information, of type 'fdel'. There must be exactly one such reference if file delivery information is needed.

It is possible that there are valid URI forms for MPEG-7 metadata (e.g. a schema URI with a fragment identifying a particular element), and it may be possible that these structures could be used for MPEG-7. However, there is explicit support for MPEG-7 in ISO base media file format family files, and this explicit support is preferred as it allows, among other things:

a) incremental update of the metadata (logically, I/P coding, in video terms) whereas this draft is 'I-frame only';

b) binarization and thus compaction;

c) the use of multiple schemas.

Therefore, the use of these structures for MPEG-7 is deprecated (and undocumented).

Information on URI forms for some metadata systems can be found in Annex G.

8.11.6.2 Syntax

```
aligned(8) class ItemInfoExtension(unsigned int(32) extension_type)
{
}
aligned(8) class FDItemInfoExtension() extends ItemInfoExtension ('fdel')
{
   string            content_location;
   string            content_MD5;
   unsigned int(64)  content_length;
   unsigned int(64)  transfer_length;
   unsigned int(8)   entry_count;
   for (i=1; i <= entry_count; i++)
      unsigned int(32)  group_id;
}

aligned(8) class ItemInfoEntry
      extends FullBox('infe', version, 0) {
   if ((version == 0) || (version == 1)) {
      unsigned int(16)  item_ID;
      unsigned int(16)  item_protection_index
      string            item_name;
      string            content_type;
      string            content_encoding;   //optional
   }
   if (version == 1) {
      unsigned int(32)  extension_type;     //optional
      ItemInfoExtension(extension_type);//optional
   }
   if (version == 2) {
      unsigned int(16)  item_ID;
      unsigned int(16)  item_protection_index;
      unsigned int(32)  item_type;

      string            item_name;
      if (item_type=='mime') {
         string         content_type;
         string         content_encoding;   //optional
      } else if (item_type == 'uri ') {
         string         item_uri_type;
      }
   }
}

aligned(8) class ItemInfoBox
      extends FullBox('iinf', version = 0, 0) {
   unsigned int(16) entry_count;
   ItemInfoEntry[ entry_count ]     item_infos;
}
```

8.11.6.3 Semantics

`item_id` contains either 0 for the primary resource (e.g., the XML contained in an `'xml '` box) or the ID of the item for which the following information is defined.
`item_protection_index` contains either 0 for an unprotected item, or the one-based index into the item protection box defining the protection applied to this item (the first box in the item protection box has the index 1).
`item_name` is a null-terminated string in UTF-8 characters containing a symbolic name of the item (source file for file delivery transmissions).
`item_type` is a 32-bit value, typically 4 printable characters, that is a defined valid item type indicator, such as 'mime'
`content_type` is a null-terminated string in UTF-8 characters with the MIME type of the item. If the item is content encoded (see below), then the content type refers to the item after content decoding.
`item_uri_type` is a string that is an absolute URI, that is used as a type indicator.
`content_encoding` is an optional null-terminated string in UTF-8 characters used to indicate that the binary file is encoded and needs to be decoded before interpreted. The values are as defined for Content-Encoding for HTTP/1.1. Some possible values are "gzip", "compress" and "deflate". An empty string indicates no content encoding. Note that the item is stored after the content encoding has been applied.
`extension_type` is a printable four-character code that identifies the extension fields of version 1 with respect to version 0 of the Item information entry.
`content_location` is a null-terminated string in UTF-8 characters containing the URI of the file as defined in HTTP/1.1 (RFC 2616).
`content_MD5` is a null-terminated string in UTF-8 characters containing an MD5 digest of the file. See HTTP/1.1 (RFC 2616) and RFC 1864.
`content_length` gives the total length (in bytes) of the (un-encoded) file.
`transfer_length` gives the total length (in bytes) of the (encoded) file. Note that transfer length is equal to content length if no content encoding is applied (see above).
`entry_count` provides a count of the number of entries in the following array.
`group_ID` indicates a file group to which the file item (source file) belongs. See 3GPP TS 26.346 for more details on file groups.

Add the following to the end of 8.11:

8.11.11 Item Data Box

8.11.11.1 Definition

Box Type: `'idat'`
Container: Metadata box (`'meta'`)
Mandatory: No
Quantity: Zero or one

This box contains the data of metadata items that use the construction method indicating that an item's data extents are stored within this box.

8.11.11.2 Syntax

```
aligned(8) class ItemDataBox extends Box('idat') {
   bit(8) data[];
}
```

8.11.11.3 Semantics

`data` is the contained meta data

8.11.12 Item Reference Box

8.11.12.1 Definition

Box Type: `'iref'`
Container: Metadata box (`'meta'`)
Mandatory: No
Quantity: Zero or one

The item reference box allows the linking of one item to others via typed references. All the references for one item of a specific type are collected into a single item type reference box, whose type is the reference type, and which has a 'from item ID' field indicating which item is linked. The items linked to are then represented by an array of 'to item ID's. All these single item type reference boxes are then collected into the item reference box. The reference types defined for the track reference box defined in 8.3.3 may be used here if appropriate, or other registered reference types.

NOTE: This design makes it fairly easy to find all the references of a specific type, or from a specific item.

8.11.12.2 Syntax

```
aligned(8) class SingleItemTypeReferenceBox(referenceType) extends
Box(referenceType) {
   unsigned int(16) from_item_ID;
   unsigned int(16) reference_count;
   for (j=0; j<reference_count; j++) {
      unsigned int(16) to_item_ID;
   }
}
aligned(8) class ItemReferenceBox extends FullBox('iref', version=0, 0) {
   SingleItemTypeReferenceBox references[];
}
```

8.11.12.3 Semantics

`reference_type` contains an indication of the type of the reference
`from_item_id` contains the ID of the item that refers to other items
`reference_count` is the number of references
`to_item_id` contains the ID of the item referred to

8.11.13 Auxiliary video metadata

An auxiliary video track used for depth or parallax information may carry a meta-data item of type 'auvd' (auxiliary video descriptor); the data of that item is exactly one si_rbsp() as specified in ISO/IEC 23002-3. (Note that si_rbsp() is externally framed, and the length is supplied by the item location information in the file format). There may be more than one of these meta-data items (e.g. one for parallax info and one for depth, in the case that the same stream serves).

In 8.13.1 Introduction, after "Pre-computed FEC reservoirs are stored as additional items in the meta box. If a source file is split into several source blocks, FEC reservoirs for each source block are stored as separate items. The relationship between FEC reservoirs and original source items is recorded in the partition entry box ('paen') located in the FD item information box ('fiin").", insert:

Pre-composed File reservoirs are stored as additional items in the container file. If a source file is split into several source blocks, each source block is stored as a separate item called a File reservoir. The relationship between File reservoirs and original source items is recorded in the partition entry box ('paen') located in the FD item information box ('fiin').

In 8.13.2.1 Definition, replace:

The FD item information box is optional, although it is mandatory for files using FD hint tracks. It provides information on the partitioning of source files and how FD hint tracks are combined into FD sessions. Each partition entry provides details on a particular file partitioning, FEC encoding and associated FEC reservoirs.

with:

The FD item information box is optional, although it is mandatory for files using FD hint tracks. It provides information on the partitioning of source files and how FD hint tracks are combined into FD sessions. Each partition entry provides details on a particular file partitioning, FEC encoding and associated File and FEC reservoirs.

Replace 8.13.2.2 (FD Item Information, Syntax) with the following:

8.13.2.2 Syntax

```
aligned(8) class PartitionEntry extends Box('paen') {
   FilePartitionBox  blocks_and_symbols;
   FECReservoirBox   FEC_symbol_locations; //optional
   FileReservoirBox  File_symbol_locations; //optional
}

aligned(8) class FDItemInformationBox
      extends FullBox('fiin', version = 0, 0) {
   unsigned int(16)  entry_count;
   PartitionEntry    partition_entries[ entry_count ];
   FDSessionGroupBox session_info;     //optional
   GroupIdToNameBox  group_id_to_name; //optional
}
```

Replace the following definitions in 8.13.3.3:

`FEC_encoding_ID` identifies the FEC encoding scheme and is subject to IANA registration (see RFC 5052). Note that i) value zero corresponds to the "Compact No-Code FEC scheme" also known as "Null-FEC" (RFC 3695); ii) value one corresponds to the “MBMS FEC” (3GPP TS 26.346); iii) for values in the range of 0 to 127, inclusive, the FEC scheme is Fully-Specified, whereas for values in the range of 128 to 255, inclusive, the FEC scheme is Under-Specified.

`FEC_instance_ID` provides a more specific identification of the FEC encoder being used for an Under-Specified FEC scheme. This value should be set to zero for Fully-Specified FEC schemes and shall be ignored when parsing a file with `FEC_encoding_ID` in the range of 0 to 127, inclusive. `FEC_instance_ID` is scoped by the `FEC_encoding_ID`. See RFC 5052 for further details.

`max_number_of_encoding_symbols` gives the maximum number of encoding symbols that can be generated for a source block for those FEC schemes in which the maximum number of encoding symbols is relevant, such as FEC encoding ID 129 defined in RFC 5052. For those FEC schemes in which the maximum number of encoding symbols is not relevant, the semantics of this field is unspecified.

`block_size` indicates the size of a block (in bytes). A block_size that is not a multiple of the encoding_symbol_length symbol size indicates with Compact No-Code FEC that the last source symbols includes padding that is not stored in the item. With MBMS FEC (3GPP TS 26.346) the padding may extend across multiple symbols but the size of padding should never be more than encoding_symbol_length.

After 8.13.6, add the following new subclause:

8.13.7 File Reservoir Box

8.13.7.1 Definition

Box Type: 'fire'
Container: Partition Entry ('paen')
Mandatory: No
Quantity: Zero or One

The File reservoir box associates the source file identified in the file partition box ('fpar') with File reservoirs stored as additional items. It contains a list that starts with the first File reservoir associated with the first source block of the source file and continues sequentially through the source blocks of the source file.

8.13.7.2 Syntax

```
aligned(8) class FileReservoirBox
      extends FullBox('fire', version = 0, 0) {
   unsigned int(16)  entry_count;
   for (i=1; i <= entry_count; i++) {
      unsigned int(16)  item_ID;
      unsigned int(32)  symbol_count;
   }
}
```

8.13.7.3 Semantics

`entry_count` gives the number of entries in the following list. An entry count here should match the total number or blocks in the corresponding file partition box.
`item_ID` indicates the location of the File reservoir associated with a source block.
`symbol_count` indicates the number of source symbols contained in the File reservoir.

Replace the first sentence of 9.1.1 (RTP and SRTP Hint Track, Introduction) as follows:

RTP is the real-time transport protocol defined by the IETF (RFC 3550 and 3551) and is currently defined to be able to carry a limited set of media types (principally audio and video) and codings.

Amend the following two paragraphs of 9.1.1 as follows:

This design decides the packet size at the time the server hint track is created; therefore, in the declarations for the hint track, we indicate the chosen packet size. This is in the sample-description. Note that it is valid for there to be several RTP hint tracks for each media track, with different packet size choices. Similarly the time-scale for the RTP clock is provided. The timescale of the server hint track is usually chosen to match the timescale of the media tracks, or a suitable value is picked for the server. In some cases, the RTP timescale is different (e.g. 90 kHz for some MPEG payloads), and this permits that variation. Session description (SAP/SDP) information is stored in user-data boxes in the track.

RTP hint tracks do not use the composition time offset table (`'ctts'`). Instead, the hinting process for server hint tracks establishes the correct transmission order and time-stamps, perhaps using the transmission time offset to set transmission times.

Replace the first sentence of 9.1.2 (Sample Description format) with:

RTP server hint tracks are hint tracks (media handler `'hint'`), with an entry-format in the sample description of `'rtp '`:

In 9.1.3, change the first paragraph as follows:

Each sample in a server hint track will generate one or more RTP packets, whose RTP timestamp is the same as the hint sample time. Therefore, all the packets made by one sample have the same timestamp. However, provision is made to ask the server to 'warp' the actual transmission times, for data-rate smoothing, for example.

Replace the definition of RTPPacket in 9.1.3.1:

```
aligned(8) class RTPpacket {
   int(32)  relative_time;
   // the next fields form initialization for the RTP
   // header (16 bits), and the bit positions correspond
   bit(2)   RTP_version;
   bit(1)   P_bit;
   bit(1)   X_bit;
   bit(4)   CSRC_count;
   bit(1)   M_bit;
   bit(7)   payload_type;
   unsigned int(16)  RTPsequenceseed;
   unsigned int(13)  reserved = 0;
   unsigned int(1)   extra_flag;
   unsigned int(1)   bframe_flag;
   unsigned int(1)   repeat_flag;
   unsigned int(16)  entrycount;
   if (extra_flag) {
      uint(32) extra_information_length;
      box   extra_data_tlv[];
   }
   dataentry   constructors[entrycount];
}
```

In 9.1.3.1:

insert the following paragraph after the code definitions and before 'In server hint tracks…':

The semantics of the fields for RTP server hint tracks is specified below. RTP reception hint tracks use the same packet structure. The semantics of the fields when the packet structure is used in an RTP reception hint track is specified in subclause 9.4.1.4.

replace:

The relative-time field 'warps' the actual transmission time away from the sample time. This allows traffic smoothing.

with:

In server hint tracks, the `relative_time` field 'warps' the actual transmission time away from the sample time. This allows traffic smoothing.

after:

The following 2 bytes exactly overlay the RTP header; they assist the server in making the RTP header.

insert the following:

Within these 2 bytes, the fields `RTP_version` and `CSRC_count` are reserved in server (transmission) hint tracks and the server fills in these fields.

replace:

The extra-flag indicates that there is extra information before the constructors, in the form of type-length-value sets. Only one such set is currently defined; `'rtpo'` gives a 32-bit signed integer offset to the actual RTP time-stamp to place in the packet. This enables packets to be placed in the hint track in decoding order, but have their presentation time-stamp in the transmitted packet be in a different order. This is necessary for some MPEG payloads. Note that the extra-information-length is the length in bytes of this field and all the TLV entries. Note also that the TLV boxes are aligned on 32-bit boundaries; the box size indicates the actual bytes used, not the padded length. The extra-information-length will be correct.

with:

`extra_flag` equal to 1 indicates that there is extra information before the constructors, in the form of type-length-value sets.

`extra_information_length` indicates the length in bytes of all extra information before the constructors, which includes the four bytes of the `extra information_length` field. The subsequent boxes before the constructors, referred to as the TLV boxes, are aligned on 32-bit boundaries. The box size of any TLV box indicates the actual bytes used, not the length required for padding to 32-bit boundaries. The value of `extra_information_length` includes the required padding for 32-bit boundaries.

`The rtpoffsetTLV ('rtpo')` gives a 32-bit signed integer offset to the actual RTP time-stamp to place in the packet. This enables packets to be placed in the hint track in decoding order, but have their presentation time-stamp in the transmitted packet be in a different order. This is necessary for some MPEG payloads.

change the last paragraph to read:

The `bframe_flag` indicates a disposable 'b-frame'. The `repeat_flag` indicates a 'repeat packet', one that is sent as a duplicate of a previous packet. Servers may wish to optimize handling of these packets.

In 9.2.1 (ALC/LCT and FLUTE Hint Track Formats, Introduction), replace the first paragraph as follows:

The file format supports multicast/broadcast delivery of files with FEC protection. Files to be delivered are stored as items in a container file (defined by the file format) and the meta box is amended with information on how the files are partitioned into source symbols. For each source block of a FEC encoding, additional parity symbols can be pre-computed and stored as FEC reservoir items. The partitioning depends on the FEC scheme, the target packet size, and the desired FEC overhead. Pre-composed source symbols can be stored as File reservoir items to minimize duplicate information in the container file especially with MBMS-FEC. The actual transmission is governed by hint tracks that contain server instructions that facilitate the encapsulation of source and FEC symbols into packets.

In 9.2.1, replace the last paragraph as follows:

The syntax for using the meta box as a container file for source files is defined in 8.11, partitions, file and FEC reservoirs are defined in 8.13, while the syntax for FD hint tracks is defined in 9.2.

Replace 9.2.4.6 as follows:

9.2.4.6 Extra Data Box

Each sample of an FD hint track may include extra data stored in an extra data box:

```
aligned(8) class ExtraDataBox extends Box('extr') {
   FECInformationBox  feci;
   bit(8)   extradata[];
}
```

After 9.2.4.6, add the following new subclause:

9.2.4.7 FEC Information Box

9.2.4.7.1 Definition

Box Type: 'feci'
Container: Extra Data Box ('extr')
Mandatory: No
Quantity: Zero or One

The FEC Information box stores FEC encoding ID, FEC instance ID and FEC payload ID which are needed when sending an FD packet.

9.2.4.7.2 Syntax

```
aligned(8) class FECInformationBox extends Box('feci') {
   unsigned int(8)   FEC_encoding_ID;
   unsigned int(16)  FEC_instance_ID;
   unsigned int(16)  source_block_number;
   unsigned int(16)  encoding_symbol_ID;
}
```

9.2.4.7.3 Semantics

`FEC_encoding_ID` identifies the FEC encoding scheme and is subject to IANA registration (see RFC 5052), in which (i) value zero corresponds to the "Compact No-Code FEC scheme" also known as "Null-FEC" (RFC 3695); (ii) value one corresponds to the "MBMS FEC" (3GPP TS 26.346); (iii) for values in the range of 0 to 127, inclusive, the FEC scheme is Fully-Specified, whereas for values in the range of 128 to 255, inclusive, the FEC scheme is Under-Specified.

`FEC_instance_ID` provides a more specific identification of the FEC encoder being used for an Under-Specified FEC scheme. This value should be set to zero for Fully-Specified FEC schemes and shall be ignored when parsing a file with FEC_encoding_ID in the range of 0 to 127, inclusive. FEC_instance_ID is scoped by the FEC_encoding_ID. See RFC 5052 for further details.

`source_block_number` identifies from which source block of the object the encoding symbol(s) in the FD packet are generated.

`encoding_symbol_ID` identifies which specific encoding symbol(s) generated from the source block are carried in the FD packet.

After 9.2, add the following new subclauses:

9.3 MPEG-2 Transport Hint Track Format

9.3.1 Introduction

MPEG-2 TS (Transport Stream) is a stream multiplex which can carry one or more programs, consisting of audio, video and other media. The file format supports the storage of MPEG-2 TS in a hint track. An MPEG-2 TS hint track can be used for both storage of received TS packets (as a reception hint track), and as a server hint track used for the generation of an MPEG-2 TS.

The MPEG-2 TS hint track definition supports so-called "precomputed hints". Precomputed hints make no use of including data by reference from other tracks, but rather MPEG-2 TS packets are stored as such. This allows reusing the MPEG-2 TS packets stored in a separate file. Furthermore, precomputed hints facilitate simple recording operation.

In addition to precomputed hint samples, it is possible to include media data by reference to media tracks into hint samples. Conversion of a received transport stream to media tracks would allow existing players compliant with earlier versions of the ISO base media file format to process recorded files as long as the media formats are also supported. Storing the original transport headers retains valuable information for error concealment and the reconstruction of the original transport stream.

9.3.2 Design Principles

The design principles of the MPEG-2 TS Hint Track Format are as follows.

A sequence of samples in an MPEG-2 TS Hint Track is a set of precomputed and constructed MPEG-2 TS packets. Precomputed packets are TS packets which are stored unchanged in the case of reception or will be sent as is. This is especially important where data cannot be de-multiplexed and elementary streams cannot be created – e.g. when the transport stream is encrypted and is not allowed to be stored decrypted. Therefore, it is necessary to be able to store the MPEG-2 TS as such in a hint track. Constructed packets use the same approach as RTP hint tracks, i.e., the sample contains instructions for a streaming server to construct the packet. The actual media data is contained in other tracks. A track reference of type 'hint' is used.

9.3.2.1 Reusing existing Transport Streams

It was desired to reuse existing TS instances and therefore an additional mechanism exists to cover a wide variety of existing TS recordings. These recordings may consist not only of TS packets but have preceding or trailing data with each TS packet. A specific case for preceding data is a 4-byte timestamp in front of each TS packet to remove the jitter of a transmission system. A specific case for trailing data is the addition of FEC when a TS packet is transmitted over an error-prone channel.

9.3.2.2 Timing

MPEG-2 TS defines a single clock for each program, running at 27MHz, which sampling value is transported as PCRs in the TS for clock recovery. The timescale of MPEG-2 TS Hint Tracks is recommended to be 90000, or an integer division or multiple thereof.

The decoding time of a sample in a MPEG-2 TS Hint Track is the reception/transmission time of the first bit of that packet or packet group which is recommended to be derived from the PCR timestamps of the TS, since if the PCR times are used, piece-wise linearity can be assumed and the 'stts' table compacts sensibly. The optional 'tsti' box in the sample description can be used to signal whether reception timing with or without clock recovery was used when the hint track is a reception hint track. In the case of a server hint track PCR timing is assumed.

> NOTE: When there are multiple packets in a sample, they cannot be given independent transmission time offsets.

9.3.2.3 Packet Grouping

The sample format for MPEG-2 Transport Stream Hint Tracks allows multiple TS packets in one sample. Specific applications, such as some IPTV applications, convey TS packets in an RTP stream. Only one reception timestamp can be derived for all TS packets carried in one RTP packet. Another application for storing multiple TS packets in a sample is SPTSs, where a sample contains all the TS packets for a GoP. In this case every sample is a random access point.

Note that random-access to every TS packet is not possible by the means of the file format if multiple TS packets per sample are used.

In the case of an MPTS only one packet per sample should be used. This facilitates the use of the sample group mechanism on a per-packet basis.

9.3.2.4 Random-access points

A random access point is a point at which processing of a track may begin without error. Both MPTS and SPTS are supported by MPEG-2 TS Hint Tracks, however a random access point, marked as a sync sample, is normally only defined for SPTS, where it specifies the beginning of a packet that contains the first byte of an independently decodable media access unit (e.g. MPEG-2 video I-frames or MPEG-4 AVC IDR pictures) of a stream that uses differential coding. For MPTS, the sync sample table would normally be present but empty, indicating that there is no point in the track at which processing of the entire track may begin without error. It is recommended that the PSI/SI be in the Sample Description so that true random-access with just the media data is possible.

Note that in the case of an MPTS, the sync sample table is present but empty (which means essentially that no sample is a sync sample).

Note also that in case of an SPTS, samples including multiple TS packets should have a sync point (e.g. GoP boundary) at the start of a sample. The sync sample table then marks the samples the sync points (e.g. the start of GoPs); if the sync sample table is absent, all the samples are sync points. If the sync sample table is present but empty, the sync sample positions are unknown and may be not at the start of samples.

NOTE: An application searching for a key frame can start reading at that location, but in general it also has to read further MPEG-2 TS packets (regarding the file format these are subsequent samples) so that the decoder can decode a complete frame.

9.3.2.5 Application as a Reception Hint Track

Reception hint tracks may be used when one or more packet streams of data are recorded. They indicate the order, reception timing, and contents of the received packets among other things.

NOTE 1: Players may reproduce the packet stream that was received based on the reception hint tracks and process the reproduced packet stream as if it was newly received.

Reception hint tracks have the same structure as hint tracks for servers.

The format of the reception hint samples is indicated by the sample description for the reception hint track. Each protocol has its own reception hint sample format and name.

NOTE 2: Servers using reception hint tracks as hints for sending of the received streams should handle the potential degradations of the received streams, such as transmission delay jitter and packet losses, gracefully and ensure that the constraints of the protocols and contained data formats are obeyed regardless of the potential degradations of the received streams.

NOTE 3: As with server hint tracks, the sample formats of reception hint tracks may enable construction of packets by pulling data out of other tracks by reference. These other tracks may be hint tracks or media tracks. The exact form of these pointers is defined by the sample format for the protocol, but in general they consist of four pieces of information: a track reference index, a sample number, an offset, and a length. Some of these may be implicit for a particular protocol. These 'pointers' always point to the actual source of the data, i.e., indirect data referencing is disallowed. If a hint track is built 'on top' of another hint track, then the second hint track must have direct references to the media track(s) used by the first where data from those media tracks is placed in the stream.

If received data is extracted to media tracks, the de-hinting process must ensure that the media streams are valid, i.e. the streams must be error-free (which requires e.g. error concealment).

9.3.3 Sample Description Format

9.3.3.1 Introduction

The sample description for an MPEG2-TS reception hint track contains all static metadata that describe the stream or a portion thereof, especially the PSI/SI tables. MPEG-2 TS reception hint tracks use an entry-format

in the sample description of 'rm2t' (which indicates *MPEG-2 Transport Stream*). The entry-format for MPEG2-TS server hint tracks is 'sm2t'.

The static metadata documents e.g. PSI/SI tables. The presence of static metadata is optional. When present, the static metadata shall be valid for the MPEG2-TS packets it describes. Consequently, if a piece of static metadata changes in the stream, a new sample entry is needed for the first sample at or after the change. If static metadata is not present in the sample entry, structures, such as PSI/SI tables, stored in the MPEG2-TS packets are valid and the stream must be scanned in order to find out which values of static metadata are valid for a particular sample.

9.3.3.2 Syntax

```
class MPEG2TSReceptionSampleEntry extends MPEG2TSSampleEntry(`rm2t´) {
}


class MPEG2TSServerSampleEntry extends MPEG2TSSampleEntry(`sm2t´) {
}


class MPEG2TSSampleEntry(name) extends HintSampleEntry(name) {
   uint(16) hinttrackversion = 1;
   uint(16) highestcompatibleversion = 1;
   uint(8)  precedingbyteslen;
   uint(8)  trailingbyteslen;
   uint(1)  precomputed_only_flag;
   uint(7)  reserved;
   box      additionaldata[];
}
```

9.3.3.3 Semantics

`hinttrackversion` is currently 1; the `highestcompatibleversion` field specifies the oldest version with which this track is backward-compatible.

`precedingbyteslen` indicates the number of bytes that are preceding each MPEG2-TS packet (which may e.g. be a time-code from an external recording device).

`trailingbyteslen` indicates the number of bytes that are at the end of each MPEG2-TS packet (which may e.g. contain checksums or other data that was added by a recording device).

`precomputed_only_flag` indicates whether the associated samples are purely precomputed if set to 1;

`additionaldata` is a set of boxes. This set can contain boxes that describe one common version of the PSI/SI tables by means of the 'tPAT' box or the 'tPMT' box or other data, e.g. boxes that are only valid for a sample (which contains multiple packets) and describe the initial conditions of the STC or boxes that define the content of the preceding or trailing data. There shall be at most one of each of PATBox, TSTimingBox, InitialSampleTimeBox present within additionaldata

The following optional boxes for `additionaldata` are defined:

```
aligned(8) class PATBox() extends Box('tPAT') {
   uint(3)     reserved;
   uint(13)    PID;
   uint(8)     sectiondata[];
}


aligned(8) class PMTBox() extends Box('tPMT') {
   uint(3)     reserved;
   uint(13)    PID;
   uint(8)     sectiondata[];
}
```

```
aligned(8) class ODBox () extends Box (‘tOD ’) {
   uint(3)      reserved;
   uint(13)     PID;
   uint(8)      sectiondata[];
}

aligned(8) class TSTimingBox() extends Box(‘tsti’) {
   uint(1)      timing_derivation_method;
   uint(2)      reserved;
   uint(13)     PID;
}

aligned(8) class InitialSampleTimeBox() extends Box(‘istm’) {
   uint(32) initialsampletime;
   uint(32) reserved;
}
```

The 'tPAT' box contains the section data of the PAT and each 'tPMT' box contains the section data of one of the PMTs.

In the case of an SPTS, it is strongly recommended that the 'tPMT' box is present in the `additionaldata`. If the PMT is not present in the sample data, then it shall be present in the `additionaldata`. If the 'tPMT' box is present, it shall be the PMT for the program contained in the sample data (although the recorded stream may contain other programs and be an MPTS).

`PID` is the PID of the MPEG2-TS packets from which the data was extracted. In the case of the 'tPAT' box this value is always 0.

`sectiondata` extends to the end of the box and is the complete MPEG2-TS table, containing the concatenated sections, of an identical version number.

`initialsampletime` specifies the initial value of the sample times in case the sample times do not start from 0. Unlike media tracks, MPEG-2 TS hint track usually have sample times not starting from 0, e.g., PCR times and reception times. Since ‘stts’ only stores the delta between sample times, this field is required for reconstructing the original sample times:

`OriginalSampleTime(n) = initialsampletime + STTS(n).`

In case PCR times are used for sample times, the reconstructed sample time can be used to initialize the STC when the sample is randomly accessed. Note that this field may need to be updated after editing.

`timing_derivation_method` is a flag which specifies the method which was used to set the sample time for a given PID. The values for `timing_derivation_method` are as follows:

`0x0` reception time: the sample timing is derived from the reception time. It is not guaranteed that the STC was recovered for derivation of the reception time.

`0x1` piecewise linearity between PCRs: the sample time is derived from a reconstructed STC for this program. Piecewise linearity between adjacent PCRs is assumed and all TS packets in the samples have a constant duration in this range.

9.3.4 Sample Format

Each sample of an MPEG-2 TS Hint track consists of a set of

- pre-computed packets: one or more MPEG-2 TS packets with the associated headers and trailers
- constructed packets: instructions to compose one or more MPEG2-TS packets with the associated headers and trailers by pointing to data of another track.

Note that each MPEG-2 TS packet in the sample may be preceded with a preheader (`precedingbytes`), or followed by a posttrailer (`trailingbytes`), as detailed in the Sample Description Format. The size of the preheader and the posttrailer are specified by `precedingbyteslen` and `trailingbyteslen`, respectively, in the sample description to allow compact sample tables with fewer chunks.

It is possible for a mixture of precomputed and constructed samples to occur in the same track. If padding of the transport stream packet is required, this can be accomplished with the `adaptation_field` or explicitly by using the `MPEG2TSImmediateConstructor` as appropriate.

NOTE: The number of MPEG-2 TS packets in the sample can be derived from the sample size table directly if the sample consists of pre-computed packets only, which is a conclusion if the `precomputed_only_flag` in the sample entry is set. The number of MPEG-2 TS packets in the sample may be variable or restricted, e.g. extensions of this file format may define a sample to contain exactly one packet.

9.3.4.1 Syntax

```
// Constructor format
aligned(8) abstract class MPEG2TSConstructor (uint(8) type) {
   uint(8)        constructor_type = type;
}

aligned(8) class MPEG2TSImmediateConstructor
   extends MPEG2TSConstructor(1) {
   uint(8)        immediatedatalen;
   uint(8)        data[immediatedatalen];
}

aligned(8) class MPEG2TSSampleConstructor
   extends MPEG2TSConstructor(2) {
   uint(8)     sampledatalen;
   uint(16)    trackrefindex;
   uint(32)    samplenumber;
   uint(32)    sampleoffset;
}

// Packet format
aligned(8) class MPEG2TSPacketRepresentation {
   uint(8)  precedingbytes[precedingbyteslen];
   uint(8)  sync_byte;
   if (sync_byte == 0x47) {
      uint(8)     packet[187];
   } else if (sync_byte == 0x00) {
      uint(8)     headerdatalen;
      uint(4)     reserved;
      uint(4)     num_constructors;
      bit(1)      transport_error_indicator;
      bit(1)      payload_unit_start_indicator;
      bit(1)      transport_priority;
      bit(13)     PID;
      bit(2)      transport_scrambling_control;
      bit(2)      adaptation_field_control;
      bit(4)      continuity_counter;
      if (adaptation_field_control == ´10´ ||
          adaptation_field_control == ´11´) {
         uint(8)     adaptation_field[headerdatalen-3];
      }
      MPEG2TSConstructor   constructors[num_constructors];
   }
   uint(8)   trailingbytes[trailingbyteslen];
}

// Sample format
aligned(8) class MPEG2TSSample {
   MPEG2TSPacket  packet[];
}
```

9.3.4.2 Semantics

`precedingbytes` contains any extra data preceding the packet, typically provided by the recording device. For example, this may include a timestamp.

`sync_byte`: if this value is 0x47, then the sample is a transport stream packet (a precomputed reception hint track sample), with the remaining bytes following in the field `packet`. If this value is 0x00, it indicates that the associated sample points to a track indexed by `trackrefindex` in the track reference box with reference type 'hint'. All other values are currently reserved. When the packet data is actually put into a streaming channel, the value shall always be set to 0x47

`trackrefindex` indexes in the track reference box with reference type 'hint' to indicate with which media track the current sample is associated. The `samplenumber` and `sampleoffset` fields in the `MPEG2TSSampleConstructor` point into this media track. The `trackrefindex` starts from value 1. The value 0 is reserved for future use.

`packet`: The MPEG-2 TS packet, apart from the sync byte (0x47).

The `MPEG2TSConstructor` array is a collection of one or more constructor entries, to allow for multiple access units in one transport stream packet. An `MPEG2TSImmediateConstructor` can contain, amongst others, the PES header. An `MPEG2TSSampleConstructor` references data in the associated media track. The sum of `headerdatalen` and the `datalen` fields of all constructors of an `MPEG2TSPacket` must be equal to the length of the transport stream packet being constructed, minus 1 byte, which is 187.

`trailingbytes` contains any extra data following the packet. For example, this may include a checksum.

`samplenumber` indicates the sample within the referred track contained in the packet and `sampleoffset` indicates the starting byte position of the referred media sample contained in the packet of which `sampledatalen` bytes are included. `sampleoffset` starts from value 0.

`immediatedatalen` indicates the number of bytes within the field `data` that are included in the sample rather than data being included into the sample by reference to a media track.

`headerdatalen` indicates the length of the TS packet header (without the sync byte) in bytes. This field has the value 3 if the `adaptation_field` is not present or the value `(adaptation_field_length+3)`, where `adaptation_field_length` is the first octet of the structure `adaptation_field` as defined in ISO/IEC 13818-1 [4].

Neither the format of `precedingbytes` nor `trailingbytes` are defined by this specification.

The remaining fields (`transport_error_indicator`, `payload_unit_start_indicator`, `transport_priority`, `PID`, `transport_scrambling_control`, `adaptation_field_control`, `continuity_counter`, `adaptation_field`) of the sample structure contain a copy of the packet header of the TS packet, as defined in ISO/IEC 13818-1 [4].

9.3.5 Protected MPEG 2 Transport Stream Hint Track

9.3.5.1 Introduction

This specification defines a mechanism for marking media streams as protected. This works by changing the four character code of the SampleEntry, and appending boxes containing both details of the protection mechanism and the original four character code. However, in this case the track is not protected; it is an 'in the clear' hint track which contains protected data. This Subclause describes how hint tracks should be marked as carrying protected data, using a similar mechanism, and utilizing the same boxes.

9.3.5.2 Syntax

```
class ProtectedMPEG2TransportStreamSampleEntry
   extends MPEG2TransportStreamSampleEntry('pm2t') {
   ProtectionSchemeInfoBox    SchemeInformation;
}
```

9.3.5.3 Semantics

The `SchemeInformation` ("sinf") box (defined in 8.12) shall contain details of the protection scheme applied. This shall include the `OriginalFormatBox` which shall contain the original sample entry type of the MPEG-2 Transport StreamSampleEntry box.

9.4 RTP, RTCP, SRTP and SRTCP Reception Hint Tracks

9.4.1 RTP Reception Hint Track

9.4.1.1 Introduction

This Subclause specifies the reception hint track format for the real-time transport protocol (RTP), as defined in IETF RFC 3550.

RTP is used for real-time media transport over the Internet Protocol. Each RTP stream carries one media type, and one RTP reception hint track carries one RTP stream. Hence, recording of an audio-visual program results into at least two RTP reception hint tracks.

The design of the RTP reception hint track format follows as much as possible the design of the RTP server hint track format. This design should ensure that RTP packet transmission operates very similarly regardless whether it is based on RTP reception hint tracks or RTP server hint tracks. Furthermore, the number of new data structures in the file format was consequently kept as small as possible.

The format of the RTP reception hint tracks allow storing of the packet payloads in the hint samples, or converting the RTP packet payloads to media samples and including them by reference to the hint samples, or combining both approaches. As noted earlier, conversion of received streams to media tracks allows existing players compliant with earlier versions of the ISO base media file format to process recorded files as long as the media formats are also supported. Storing the original RTP headers retains valuable information for error concealment and the reconstruction of the original RTP stream. It is noted that the conversion of packet payloads to media samples may happen "off-line" after recording of the streams in precomputed RTP reception hint tracks has been completed.

9.4.1.2 Sample Description Format

The entry-format in the sample description for the RTP reception hint tracks is 'rrtp'. The syntax of the sample entry is the same as for RTP server hint tracks having the entry-format 'rtp '.

```
class ReceivedRtpHintSampleEntry() extends SampleEntry (‘rrtp‘) {
   uint(16)    hinttrackversion = 1;
   uint(16)    highestcompatibleversion = 1;
   uint(32)    maxpacketsize;
   box         additionaldata[];
}
```

The entry-format identifier in the sample description of the RTP reception hint track is different from the entry-format in the sample description of the RTP server hint track, in order to avoid using an RTP reception hint track that contains errors as a valid server hint track.

The `additionaldata` set of boxes may include the timescale entry ('tims') and time offset ('tsro') boxes. Moreover, the `additionaldata` may contain a timestamp synchrony box.

The timescale entry box (`‘tims’`) shall be present and the value of timescale shall be set to match the clock frequency of the RTP timestamps of the stream captured in the reception hint track.

The time offset box (`‘tsro’`) may be present. If the time offset box is not present, the value of the field `offset` is inferred to be equal to 0. The value of the field `offset` is used for the derivation of the RTP timestamp, as specified in 9.4.1.4.

RTP timestamps typically do not start from zero, especially if an RTP receiver 'tunes' into a stream. The time offset box should therefore be present in RTP reception hint tracks and the value of `offset` in the time offset box should be set equal to the first RTP timestamp of the RTP stream in reception order.

Zero or one `timestampsynchrony` boxes may be present in the `additionaldata` of the sample entry for a RTP reception hint track. If a `timestampsynchrony` box is not present, the value of `timestamp_sync` is inferred to be equal to 0.

```
class timestampsynchrony() extends Box('tssy') {
   unsigned int(6) reserved;
   unsigned int(2) timestamp_sync;
}
```

`timestamp_sync` equal to 0 indicates that the RTP timestamps of the present RTP reception hint track derived from the equation in 9.4.1.4 may or may not be synchronized with RTP timestamps of other RTP reception hint tracks.

`timestamp_sync` equal to 1 indicates that the RTP timestamps of the present RTP reception hint track derived from the equation in 9.4.1.4 reflect the received RTP timestamps exactly (without corrected synchronization to any other RTP reception hint track).

`timestamp_sync` equal to 2 indicates that RTP timestamps of the present RTP reception hint track derived from the equation in 9.4.1.4 are synchronized with RTP timestamps of other RTP reception hint tracks.

When `timestamp_sync` is equal to 0 or 1, a player should correct the inter-stream synchronization using stored RTCP sender reports. When `timestamp_sync` is equal to 2, the media contained in the RTP reception hint tracks can be played out synchronously according to the reconstructed RTP timestamps without synchronization correction using RTCP Sender Reports. If it is expected that the RTP reception hint track will be used for re-sending the recorded RTP stream, it is recommended that `timestamp_sync` be set equal to 0 or 1, because the stored RTCP sender reports can be reused.

`timestamp_sync` equal to 3 is reserved.

The value of `timestamp_sync` shall be identical for all RTP reception hint tracks present in a file.

When RTCP is also stored, using an RTCP hint track, the timestamp relationship between the RTP and RTCP hint tracks can only be maintained if the RTP timestamps are anchored by using a set time offset ('tsro') in the RTP track, and hence the time offset is mandatory if RTCP is stored in an RTCP hint track.

Zero or one `ReceivedSsrcBox` identified with the four-character code 'rssr' shall be present in the `additionaldata` of a sample descriptor entry of a RTP reception hint track:

```
class ReceivedSsrcBox extends Box('rssr') {
   unsigned int(32) SSRC
}
```

The `SSRC` value must equal the SSRC value in the header of all recorded SRTP packets described by the sample description.

9.4.1.3 Sample Format

The sample format of RTP reception hint tracks is identical to the syntax of the sample format of the RTP server hint tracks. Each sample in the reception hint track represents one or more received RTP packets. If media frames are not both fragmented and interleaved in an RTP stream, it is recommended that each sample represents all received RTP packets that have the same RTP timestamp, i.e., consecutive packets in RTP sequence number order with a common RTP timestamp.

Each RTP reception hint sample contains two areas: the instructions to compose the packet, and any extra data needed for composing the packet, such as a copy of the packet payload. Note that the size of the sample is known from the sample size table.

Since the reception time for the packets may vary, this variation can be signalled for each packet as specified subsequently.

9.4.1.4 Packet Entry Format

Each packet in the packet entry table has same structure as for server (transmission) hint tracks, in 9.1.3.1.

Where `i` is the sample number of a sample, the sum of the sample time DT(i) as specified in 8.6.1.2 and `relative_time` indicates the reception time of the packet. The clock source for the reception time is undefined and may be, for instance, the wall clock of the receiver. If the range of reception times of a reception hint track overlaps entirely or partly with the range of reception times of another reception hint track, the clock sources for these hint tracks shall be the same.

It is recommended that receivers may use a constant value for `sample_delta` in the decoding time to sample box ('stts') as much as reasonable and smooth out packet scheduling and end-to-end delay variation by setting `relative_time` adaptively in stored reception hint samples. This arrangement of setting the values of `sample_delta` and `relative_time` can facilitate a compact decoding time to sample box. In this case timestamp_sync is set to 1, the sample durations are mostly constant, and the time offset ('tsro') is stored in the sample entry.

The values of `RTP_version`, `P_bit`, `X_bit`, `CSRC_count`, `M_bit`, `payload_type`, and `RTPsequenceseed` shall be set equal to the V, P, X, CC, M, PT and sequence number fields of the RTP packet captured in the sample.

The fields `bframe_flag` and `repeat_flag` are reserved in reception hint tracks and must be zero.

The semantics of `extra_flag` and `extra_information_length` are identical to those of specified for the RTP server hint tracks.

The following TLV boxes are specified: `rtphdrextTLV`, `rtpoffsetTLV`, `receivedCSRC`.

If the `X_bit` is set a single `rtphdrextTLV` box shall be present for storing the received RTP Header Extension.

```
aligned(8) class rtphdrextTLV extends Box('rtpx') {
   unsigned int(8) data[];
}
```

`data` is the raw RTP Header Extension which is application-specific.

The syntax of the `rtpoffsetTLV` box is specified in 9.1.3.1

`offset` indicates a 32-bit signed integer offset to the RTP timestamp of the received RTP packet. Let i be the sample number of a sample, DT(i) be equal to DT as specified in 8.6.1.2 for sample number i, `tsro.offset` be the value of offset in the 'tsro' box of the referred reception hint sample entry, and % be the modulo operation. The value of `offset` shall be such that the following equation is true:

$$RTPtimestamp = (DT_i + tsro.offset + offset) \bmod 2^{32}$$

Equation 1: RTP timestamp calculation

NOTE 1: When each reception hint sample represents all received RTP packets that have the same RTP timestamp, the value of `sample_delta` in the decoding time to sample box can be set to match the RTP timestamp. In other words, DT(i), as specified above, can be set equal to (the RTP timestamp – `tsro.offset` – `offset`) (assuming that the resulting value would be greater than or equal to 0). This is recommended.

NOTE 2: RTP timestamps do not necessarily increase as a function of RTP sequence number in all RTP streams, i.e., transmission order and playback order of packets may not be identical. For example, many video

coding schemes allow bi-prediction from previous and succeeding pictures in playback order. As samples appear in tracks in their decoding order, i.e., in reception order in case of RTP reception hint tracks, `offset` in the `rtpoffsetTLV` box can be used to warp the RTP timestamp away from the sample time DT(i).

For the purpose of edits in Edit List Boxes, the composition time of a received RTP packet is inferred to be the sum of the sample time DT(i) and `offset` as specified above.

If the value of `CSRC_count` is not equal to zero, a `receivedCSRC` box may be present for storing the received CSRC header fields for each RTP packet. The `receivedCSRC` box is identified with the four-character code 'rcsr'

```
aligned(8) class receivedCSRC extends Box('rcsr') {
   unsigned int(32)  CSRC[];  //to end of the box
}
```

The number of entries in `CSRC[]` equals the `CC` value of received SRTP packets. The n^{th} entry of `CSRC[]` shall equal the n^{th} CSRC value of the RTP packet header.

9.4.1.5 SDP information

Both movie and track SDP information may be present, as specified in 9.1.4.

9.4.2 RTCP Reception Hint Track

9.4.2.1 Introduction

This Subclause specifies the reception hint track format for the real-time control protocol (RTCP), defined in IETF RFC 3550.

RTCP is used for real-time transport of control information for an RTP session over the Internet Protocol. During streaming, each RTP stream typically has an accompanying RTCP stream that carries control information for the RTP stream. One RTCP reception hint track carries one RTCP stream and is associated to the corresponding RTP reception hint track through a track reference.

The format of the RTCP reception hint tracks allows the storage of RTCP Sender Reports in the hint samples.

The RTCP Sender Reports are of particular interest for stream recording, because they reflect the current status of the server, e.g., the relationship of the media timing (RTP timestamp of audio/video packets) to the server time (absolute time in NTP format). Knowledge of this relationship is also necessary for playback of recorded RTP reception hint tracks to be able to detect and correct clock drift and jitter.

The timestamp synchrony box as specified in 9.4.1.2 makes it possible to correct clock drift and jitter before playing a file, and therefore recording of RTCP streams is optional when timestamp_sync is equal to 2.

There is no server hint track equivalent for the RTCP reception hint track, since RTCP messages are generated on-the-fly during transmission.

9.4.2.2 General

There shall be zero or one RTCP reception hint track for each RTP reception hint track. An RTCP reception hint track shall contain a track reference box including a reference of type '`cdsc`' to the associated RTP reception hint track.

When i is the sample number of a sample, the sample time DT(i) as specified in 8.6.1.2 indicates the reception time of the packet. The clock source for the reception time shall be the same as for the associated RTP reception hint track. The value of `timescale` in the Media Header Box of an RTCP reception hint track shall be equal to the value of `timescale` in the media header box of the associated RTP reception hint track.

9.4.2.3 Sample Description Format

The entry-format in the sample description for the RTCP reception hint tracks is 'rtcp'. It is otherwise identical in structure to the sample entry format for RTP. There are no defined boxes for the `additionaldata` field.

9.4.2.4 Sample Format

9.4.2.4.1 Introduction

Each sample in the reception hint track represents one or more received RTCP packets. Each sample contains two areas: the raw RTCP packets and any extra data needed. Note that the size of the sample is known from the sample size table, and that the size of an RTCP packet is indicated within the packet itself (as documented in RFC 3550), as a count one less than the number of 32-bit words in that packet.

9.4.2.4.2 Syntax

```
aligned(8) class receivedRTCPpacket {
   unsigned int(8)      data[];
}

aligned(8) class receivedRTCPsample {
   unsigned int(16)  packetcount;
   unsigned int(16)  reserved;
   receivedRTCPpacket   packets[packetcount];
}
```

9.4.2.4.3 Semantics

`data` contains a raw RTCP packet including the RTCP report header, the 20-byte sender information block and any number of report blocks. Note that the size of each RTCP packet is known by parsing the 16-bit length field of the RTCP header.
`packetcount` indicates the number of received RTCP packets contained in the sample.
`packets` contains the received RTCP packets.

9.4.3 SRTP Reception Hint Track

9.4.3.1 Introduction

This Subclause specifies the reception hint track formats for the secure real-time transport protocol (SRTP), as defined in IETF RFC 3711.

SRTP is a secure extension of the real-time media transport (RTP) over the Internet Protocol. Each SRTP stream carries one media type, and one SRTP reception hint track carries one SRTP stream. Hence, recording of an audio-visual program results into at least two SRTP reception hint tracks.

The design of the SRTP reception hint track format follows the design of RTP reception hint tracks and reuses most of the framework provided by RTP reception hint tracks. The major difference between RTP and SRTP reception hint tracks is that the actual media payload is stored in an encrypted form for SRTP reception hint tracks, whereas it is unencrypted for RTP reception hint tracks. SRTP reception hint tracks provide additional boxes to store information necessary to decrypt encrypted content on playback. Additionally, all header fields of the SRTP packet header shall be stored with the payload, as this information is necessary to check the integrity of the received data. SRTP reception hint tracks are commonly used together with SRTCP reception hint tracks.

SRTP reception hint tracks may, for example, be used to store protected mobile TV content.

9.4.3.2 Sample Description Format

9.4.3.2.1 Sample Description Entry

The sample description format for SRTP reception hint tracks is identical to that for RTP reception hint tracks with the exception that the sample entry name is changed from 'rrtp' to 'rsrp' and that it may contain additional boxes:

```
class ReceivedSrtpHintSampleEntry() extends SampleEntry ('rsrp') {
   uint(16)    hinttrackversion = 1;
   uint(16)    highestcompatibleversion = 1;
   uint(32)    maxpacketsize;
   box         additionaldata[];
}
```

Fields and boxes are identical to those of the `ReceivedRtpHintSampleEntry ('rrtp')`. The `addtionaldata[]` of each sample description entry of a SRTP Reception Hint Track shall contain exactly one `ReceivedSsrc` Box ('`rssr`').

Additionally, the additionaldata[] may contain the Received Cryptographic Context ID box and the Rollover Counter box defined below. Furthermore, a SRTP Process Box shall also be included as one of the `additionaldata` boxes. As the content is stored encrypted, the integrity and the encryption algorithm fields in the SRTP Process box specify the algorithm that was applied to the received stream. An entry of four spaces ($20$20$20$20) may be used to indicate that the algorithm is defined by means outside the scope of this document.

9.4.3.2.2 Received Cryptographic Context ID Box

Zero or one `ReceivedCryptoContextIdBox`, identified with the four-character code 'ccid', may be present in the `additionaldata` of a sample descriptor entry of an SRTP reception hint track. Information to recover the cryptographic context for the received SRTP stream may be stored here.

```
aligned(8) class ReceivedCryptoContextIdBox extends Box ('ccid') {
   unsigned int(16)  destPort;
   unsigned int(8) ip_version;
   switch (ip_version) {
      case 4: // IPv4
         unsigned int(32)  destIP;
         break;
      case 6: // IPv6
         unsigned int(64)  destIP;
         break;
   }
}
```

The `destPort` and `destIP` parameters contain the port number and the IP address (as present in the received IPv4 or IPv6 packages), respectively, of the SRTP session via which the recorded SRTP packets were received. `ip_version` contains either 4 or 6 representing IPv4 or IPv6, respectively.

9.4.3.2.3 Rollover Counter Box

Zero or one `RolloverCounterBox`, identified with the four-character code 'sroc', may be present in the `additionaldata` of a sample descriptor entry of an SRTP reception hint track. Typically, the rollover counter value changes every 65536 SRTP package.

```
aligned(8) class RolloverCounterBox extends Box ('sroc') {
   unsigned int(32)  rollover_counter;
   }
```

The `rollover_counter` is a non-zero integer that gives the value of the ROC field for all associated received SRTP packets.

NOTE: The rollover counter (ROC) is an element of the cryptographic context of a SRTP stream and depends on the absolute position of a packet in an RTP stream. Knowledge of the ROC value is necessary in order to decrypt a received SRTP packet. It is optional to use the rollover counter box as RFC 4771 defines as an optional mechanism to signal the ROC value explicitly in the authentication tag of a SRTP package.

9.4.3.3 Sample and Packet Entry Format

Both, sample format and packet Entry format for SRTP reception hint tracks are identical to those of RTP reception hint tracks, defined in 9.4.1.3 and 9.4.1.4. The packet payload is stored as received in the SRTP packets, i.e., all information received in the SRTP packet excluding the header or, in other words, the encrypted payload together with the key identifier (MKI) and the authentication tag.

If the value of `CSRC_count` is not equal to zero for a received SRTP packet, the `extra_data_tlv` corresponding to this `receivedSRTPpacket` shall contain exactly one `receivedCSRC` box (`'rcsr'`).

9.4.4 SRTCP Reception Hint Tracks

9.4.4.1 Introduction

This Subclause specifies the reception hint track format for the secure real-time control protocol (SRTCP), defined in IETF RFC 3711.

SRTCP is used for real-time transport of control information for a SRTP session over the Internet Protocol. SRTCP takes for SRTP the role that RTCP takes for RTP, cf., 9.4.2. During streaming, each SRTP stream typically has an accompanying SRTCP stream that carries control information for the SRTP stream. One SRTCP reception hint track carries one SRTCP stream and is associated to the corresponding SRTP reception hint track through a track reference.

The format of the SRTCP reception hint tracks allows the storage of SRTCP Packets in the hint samples, e.g., of SRTCP Sender Reports.

The SRTCP Sender Reports are of particular interest for stream recording, because they reflect the current status of the server, e.g., the relationship of the media timing (SRTP timestamp of audio/video packets) to the server time (absolute time in NTP format). Knowledge of this relationship is also necessary for playback of recorded SRTP reception hint tracks in order to be able to detect and correct clock drift and jitter.

The timestamp synchrony box as specified in 9.4.1.2 makes it possible to correct clock drift and jitter before playing a file, and therefore recording of SRTCP streams is optional.

There is no server hint track equivalent for the SRCTP reception hint track, since SRTCP messages are generated on-the-fly during transmission.

9.4.4.2 General

There shall be zero or one SRTCP reception hint track for each SRTP reception hint track. An SRTCP reception hint track shall contain a track reference box including a reference of type `'cdsc'` to the associated SRTP reception hint track.

When i is the sample number a sample, the sample time DT(i) as specified in 8.6.1.2 indicates the reception time of the packet. The clock source for the reception time shall be the same as for the associated SRTP reception hint track. The value of `timescale` in the Media Header Box of an SRTCP reception hint track shall be equal to the value of `timescale` in the media header box of the associated SRTP reception hint track.

9.4.4.3 Sample Description Format

The entry-format in the sample description for the SRTCP reception hint tracks is 'stcp'. It is otherwise identical in structure to the sample entry format for RTCP. The encryption and authentication method of the

SRTCP hint tracks are defined by the respective entries in SRTP Process box of the corresponding SRTP hint track.

NOTE: An equivalent to the ROC boxes defined for SRTP is not necessary for SRTCP, as the SRTCP packet contains an explicitly signalled initialization vector.

9.4.4.4 Sample Format

Sample format is the sample format for RTCP reception hint tracks as defined in 9.4.2.4

9.4.5 Protected RTP Reception Hint Track

9.4.5.1 Introduction

This specification defines a mechanism for marking media streams as protected. This works by changing the four character code of the SampleEntry, and appending boxes containing both details of the protection mechanism and the original four character code. However, in this case the track is not protected; it is an 'in the clear' hint track which contains protected data. This Subclause describes the how reception hint tracks should be marked as carrying protected data, using a similar mechanism, and utilizing the same boxes.

9.4.5.2 Syntax

```
Class ProtectedRtpReceptionHintSampleEntry
   extends RtpReceptionHintSampleEntry ('prtp') {
   ProtectionSchemeInfoBox    SchemeInformation;
}
```

9.4.5.3 Semantics

The `SchemeInformation` ('sinf') box shall contain details of the protection scheme applied. This shall include the `OriginalFormatBox` which shall contain the four character code 'rrtp' (the four character code of the original RTPReceptionHintSampleEntry box).

9.4.6 Recording Procedure

Recording of RTP streams can result into three basic file structures.

- A file containing only RTP reception hint tracks: No media tracks are included. This file structure enables efficient processing of transmission errors, but only players capable of parsing RTP reception hint track can play the file.
- A file containing only media tracks: No RTP reception hint tracks are included. This file structure allows existing players compliant with the ISO base media file format process recorded files as long as the media formats are also supported. However, sophisticated processing of transmission errors is not possible due to reasons explained in subsequent clauses.
- A file containing both RTP reception hint tracks and media tracks: This file structure has both the benefits mentioned above.

Some implementations may record first to RTP reception hint tracks only and create a file with a combination of media tracks and RTP reception hint tracks off-line.

The file structures and the operation of a recording unit are described here without reference to movie fragments. However, movie fragments can be used in files containing recorded RTP streams. For example, when streams are stored to and played from the same file simultaneously, movie fragments are necessary. Furthermore, movie fragments can be used to control the consumption of the operating memory used for maintaining the data structures for the movie Box.

9.4.6.1 Creation of a File with RTP Reception Hint Tracks Only

RTP reception hint tracks enable detection of packet losses. Consequently, parsers capable of processing RTP reception hint tracks may handle packet losses more gracefully compared to the playback of media tracks generated from received streams.

A recording operation creating one reception hint sample, i.e., one `receivedRTPsample` structure, per each RTP packet is described next. The `receivedRTPsample` structure is set to contain one `receivedRTPpacket` containing one packet constructor of type 2. The precomputed packet payload is copied to the `extradata` section of the sample. The track reference of the constructor is set to point to the hint track itself, i.e., is set equal to -1, and `sampleoffset` and `length` are set to match to the location and size of the packet payload.

If media frames are not both fragmented and interleaved in an RTP stream, it is recommended that each sample represents all received RTP packets that have the same RTP timestamp, i.e., consecutive packets in RTP sequence number order with a common RTP timestamp. The recording operation for storing all packets sharing the same RTP timestamps as one precomputed reception hint sample is very similar to the one described above. The RTP payloads (without the RTP Header Extension) sharing the same RTP timestamp are stored sequentially to `extradata` of the `receivedRTPsample` structure.

Depending on the capabilities of the implementation the streams may be recorded and be either synchronous or contain an associated RTCP reception hint track.

9.4.6.2 Creation of a File with Media Tracks Only

Media samples are created from the received RTP packets as instructed by the relevant RTP payload specification and the RTP specification. However, most media coding standards only specify the decoding of error-free streams, and consequently it should be ensured that the content in media tracks can be correctly decoded by any standard-compliant media decoder. Handling of transmission errors therefore requires two steps: detection of transmission errors and inference of samples that can be decoded correctly. These steps are described in the subsequent paragraphs.

Lost RTP packets can be detected from a gap in RTP sequence number values. RTP packets containing bit errors are usually not forwarded to the application, as their UDP checksum fails and packets are discarded in the protocol stack of the receiver. Consequently, bit-erroneous packets are usually treated as packet losses in the receiver.

The inference of media samples that can be correctly decoded depends on the media coding format and is therefore not described here in details. Generally, inter-sample prediction is weak or non-existing in audio coding formats, whereas video coding formats utilize inter prediction heavily. Consequently, a lost sample in many audio formats can often be replaced by a silent or error-concealed audio sample. It should be analyzed whether a loss of a video packet concerned a non-reference picture or a reference picture, or, more generally, in which level of the temporal scalability hierarchy the loss occurred. It should then be concluded which pictures may not be correctly decodable. For example, a loss of a non-reference picture does not affect the decoding of any other pictures, whereas a loss of a reference picture in the base temporal level typically affects all pictures until the next picture for random access, such as an IDR picture in H.264/AVC. Video tracks must not contain any samples dependent on any lost video sample.

Synchronous media streams can be achieved by parsing the incoming RTCP stream in addition to the RTP stream. This procedure is defined by the RTP specification.

9.4.6.3 Creation of a File with Both RTP Reception Hint Tracks and Media Tracks

Media samples and tracks are created from the received RTP packets as explained in 9.4.6.2. RTP reception hint tracks are created as explained in 9.4.6.1, but the creation of `receivedRTPpacket` is conditional on the existence of the corresponding media sample as follows.

- If the packet payload of the received RTP packet is represented in a media track, the track reference of the relevant packet constructors are set to point to the media track and include the packet payload

by reference. It is not recommended to have a copy of the packet payload in the `extradata` section of the received RTP sample in order to save storage space and make file editing operations easier to implement.

- If the packet payload of the received RTP packet is not represented in a media track, the instance of the `receivedRTPpacket` structure is created as explained in 9.4.6.1.

Associated RTCP reception hint tracks can be used to recover the timing relations of several RTP reception hint tracks so that the media streams are in synch. This is not necessary if the RTP reception hint tracks are marked to be in synch already by the timestamp synchrony box ('tssy').

9.4.7 Parsing Procedure

File players should prefer an RTP reception hint track over any other tracks in the same alternate group, indicated by the same value of `alternate_group` in the track header box. File players may re-create the packet stream that was received based on the reception hint tracks and process the re-created packet stream as if it was newly received. The reaction to packet losses depends on the particular media decoder implementation and may also depend on user preferences. Hence, no guidelines on handling of transmission errors are given here. However, at minimum, file players should play the media track in the same alternate group as the RTP reception hint track.

After 10.2, add the following new subclause:

10.3 Alternative Startup Sequences

10.3.1 Definition

An alternative startup sequence contains a subset of samples of a track within a certain period starting from a sync sample. By decoding this subset of samples, the rendering of the samples can be started earlier than in the case when all samples are decoded.

An `'alst'` sample group description entry indicates the number of samples in any of the respective alternative startup sequences, after which all samples should be processed.

Either version 0 or version 1 of the Sample to Group Box may be used with the alternative startup sequence sample grouping. If version 1 of the Sample to Group Box is used, `grouping_type_parameter` has no defined semantics but the same algorithm to derive alternative startup sequences should be used consistently for a particular value of `grouping_type_parameter`.

A player utilizing alternative startup sequences could operate as follows. First, a sync sample from which to start decoding is identified by using the Sync Sample Box. Then, if the sync sample is associated to a sample group description entry of type `'alst'` where `roll_count` is greater than 0, the player can use the alternative startup sequence. The player then decodes only those samples that are mapped to the alternative startup sequence until the number of samples that have been decoded is equal to `roll_count`. After that, all samples are decoded.

10.3.2 Syntax

```
class AlternativeStartupEntry() extends VisualSampleGroupEntry ('alst')
{
  unsigned int(16) roll_count;
  unsigned int(16) first_output_sample;
  for (i=1; i <= roll_count; i++)
    unsigned int(32) sample_offset[i];
  j=1;
  do { // optional, until the end of the structure
    unsigned int(16) num_output_samples[j];
    unsigned int(16) num_total_samples[j];
    j++;
  }
}
```

10.3.3 Semantics

`roll_count` indicates the number of samples in the alternative startup sequence. If `roll_count` is equal to 0, the associated sample does not belong to any alternative startup sequence and the semantics of `first_output_sample` are unspecified. The number of samples mapped to this sample group entry per one alternative startup sequence shall be equal to `roll_count`.

`first_output_sample` indicates the index of the first sample intended for output among the samples in the alternative startup sequence. The index is of the sync sample starting the alternative startup sequence is 1, and the index is incremented by 1, in decoding order, per each sample in the alternative startup sequence.

`sample_offset[i]` indicates the decoding time delta of the i-th sample in the alternative startup sequence relative to the regular decoding time of the sample derived from the Decoding Time to Sample Box or the Track Fragment Header Box. The sync sample starting the alternative startup sequence is its first sample.

`num_output_samples[j]` and `num_total_samples[j]` indicate the sample output rate within the alternative startup sequence. The alternative startup sequence is divided into k consecutive pieces, where each piece has a constant sample output rate which is unequal to that of the adjacent pieces. The first piece starts from the sample indicated by `first_output_sample`. `num_output_samples[j]` indicates the number of the output samples of the j-th piece of the alternative startup sequence. `num_total_samples[j]` indicates the total number of samples, including those that are not in the alternative startup sequence, from the first sample in the j-th piece that is output to the earlier one (in composition order) of the sample that ends the alternative startup sequence and the sample that immediately precedes the first output sample of the (j+1)th piece.

10.3.4 Examples

Hierarchical temporal scalability (e.g., in AVC and SVC) improves compression efficiency but increases the decoding delay due to reordering of the decoded pictures from the (de)coding order to output order. Deep temporal hierarchies have been demonstrated to useful in terms of compression efficiency in some studies. When the temporal hierarchy is deep and the operation speed of the decoder is limited (to no faster than real-time processing), the initial delay from the start of the decoding to the start of rendering is substantial and may affect the end-user experience negatively.

Figure AMD1.1 illustrates a typical hierarchically scalable bitstream with five temporal levels. Figure AMD1.1a shows the example sequence in output order. Values enclosed in boxes indicate the frame_num value of the picture. Values in italics indicate a non-reference picture while the other pictures are reference pictures. Figure AMD1.1b shows the example sequence in decoding order. Figure AMD1.1c shows the example sequence in output order when assuming that the output timeline coincides with that of the decoding timeline and the decoding of one picture lasts one picture interval. It can be seen that playback of the stream starts five picture intervals later than the decoding of the stream started. If the pictures were sampled at 25 Hz, the picture interval is 40 msec, and the playback is delayed by 0.2 sec.

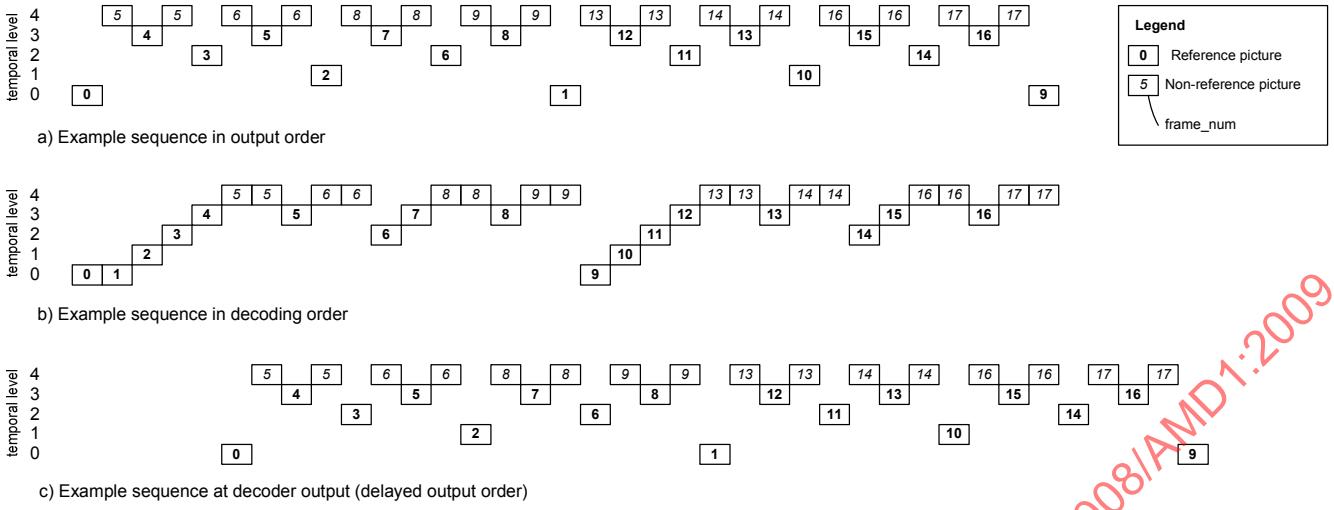

Figure AMD1.1 — Decoded picture buffering delay of an example sequence with five temporal levels.

Thanks to the temporal hierarchy, it is possible to decode only a subset of the pictures at the beginning of the sequence. Consequently, rendering can be started faster but the displayed picture rate is lower at the beginning. In other words, a player can make a trade-off between the duration of the initial startup delay and the initial displayed picture rate. Figure AMD1.2 and Figure AMD1.3 show two examples of alternative startup sequences where a subset of the bitstream of Figure AMD1.1 is decoded.

The samples selected for decoding and the decoder output are presented in Figure AMD1.2a and Figure AMD1.2b, respectively. The reference picture having frame_num equal to 4 and the non-reference pictures having frame_num equal to 5 are not decoded. In this example, the rendering of pictures starts four picture intervals earlier than in Figure AMD1.1. When the picture rate is 25 Hz, the saving in startup delay is 160 msec. The saving in the startup delay comes with the disadvantage of a lower displayed picture rate at the beginning of the bitstream.

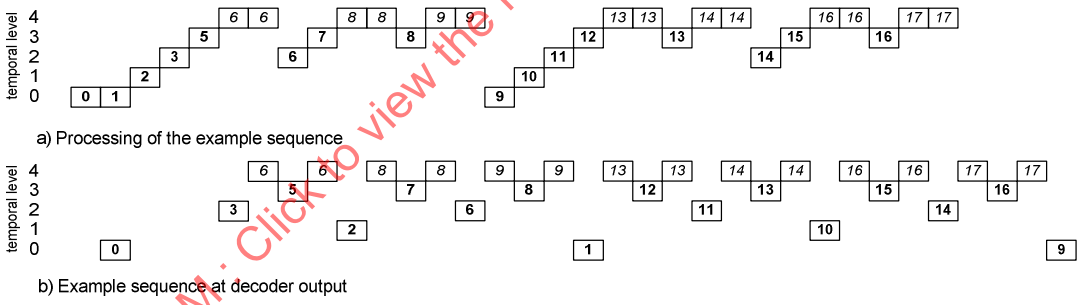

Figure AMD1.2 — An example of an alternative startup sequence.

In the example of Figure AMD1.3, another way of selecting the pictures for decoding is presented. The decoding of the pictures that depend on the picture with frame_num equal to 3 is omitted and the decoding of non-reference pictures within the second half of the first group of pictures is omitted too. The decoded picture resulting from the sample with frame_num equal to 2 is the first one that is output. As a result, the output picture rate of the first group of pictures is half of normal picture rate, but the display process starts two frame intervals (80 msec in 25 Hz picture rate) earlier than in the conventional solution illustrated in Figure AMD1.1.

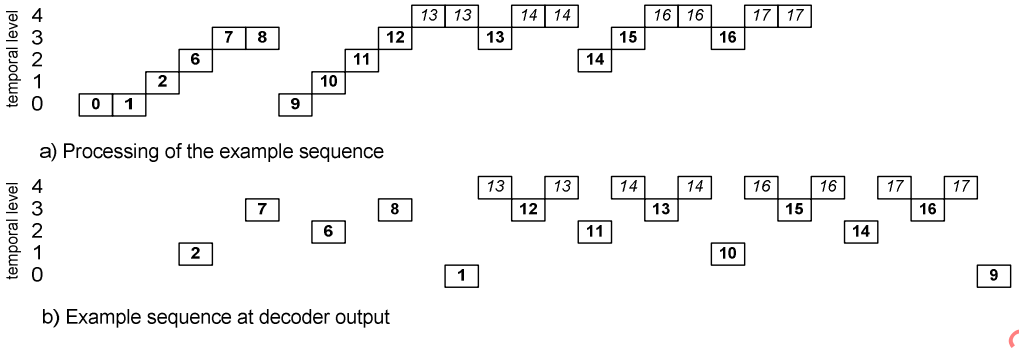

Figure AMD1.3 — Another example of an alternative startup sequence.

Replace the content of Annex C with the following:

C.1 Introduction

This Annex provides informative text to explain how to derive a specific file format from the ISO Base Media File Format.

ISO/IEC 14496-12 | ISO/IEC 15444-12 ISO Base Media Format defines the basic structure of the file format. Media-specific and user-defined extensions can be provided in other specifications that are derived from the ISO Base Media File Format.

C.2 Principles

C.2.1 General

A number of existing file formats use the ISO Base Media File Format, not least the MPEG-4 MP4 File Format (ISO/IEC 14496-14), and the Motion JPEG 2000 MJ2 File Format (ISO/IEC 15444-3). When considering a new specification derived from the ISO Base Media File format, all the existing specifications should be used both as examples and a source of definitions and technology. Check with the registration authority to find what might already exist, and what specifications exist.

In particular, if an existing specification already covers how a particular media type is stored in the file format (e.g. MPEG-4 video in MP4), that definition should be used and a new one should not be invented. In this way specifications which share technology will also share the definition of how that technology is represented.

Be as permissive as possible with respect to the presence of other information in the file; indicate that unrecognized boxes and media may be ignored (not “should be ignored”). This permits the creation of hybrid files, drawing from more than one specification, and the creation of multi-format players, capable of handling more than one specification.

When layering on this specification, it's worth observing that there are some characteristics that are intentionally ‘parameters’ to the lower (Part 12) specification, that need to be specified. Equally, there are some characteristics of the Part 12 file format specification that are internal and should rarely be discussed by other specifications. Of course, there are some characteristics in a grey area in between.

Derived specifications are ideally written solely in terms of the parameters of the Part 12 file format; what a sample is, what its timestamps mean, and so on. Mentioning specific existing boxes in a derived specification may often turn out to be an error, except in limited cases (e.g. adding a user-data box, or an extension box).