INTERNATIONAL STANDARD

ISO/IEC 14496-12

Seventh edition
2022-01

Information technology — Coding of audio-visual objects —

Part 12: **ISO base media file format**

Technologies de l'information — Codage des objets audiovisuels — Partie 12: Format ISO de base pour les fichiers médias

Reference number
ISO/IEC 14496-12:2022(E)

Contents

Page

Foreword x
Introduction xi
1 Scope 1
2 Normative references 1
3 Terms, definitions and abbreviated terms 2
3.1 Terms and definitions 2
3.2 Abbreviated terms 7
4 Object-structured file organization 7
4.1 File structure 7
4.2 Object structure 7
4.2.1 Object syntax conventions 7
4.2.2 Object definitions 8
4.2.3 Extensibility of object definitions 9
4.3 File-type box 10
4.3.1 Definition 10
4.3.2 Syntax 10
4.3.3 Semantics 11
4.4 Extended type box 11
4.4.1 Definition 11
4.4.2 Syntax 11
4.4.3 Semantics 11
5 Structure of this document 12
6 ISO base media file organization 12
6.1 Files, segments, and streams 12
6.2 Presentation structure 13
6.2.1 Object structure of a presentation 13
6.2.2 Meta data and media data 13
6.3 Structure-data (objects) 13
6.3.1 Box 13
6.3.2 Data types and fields 13
6.3.3 URIs as type indicators 14
6.3.4 Box order 15
6.4 Time structure overview 18
6.5 Identifiers 19
6.6 Brand identification 19
6.7 Uniform resource locators (URLs) 19
7 Streaming support 19
8 Box structures 19
8.1 File structure and general boxes 19
8.1.1 Media data box 19
8.1.2 Free space box 20
8.1.3 Progressive download information box 20
8.1.4 Identified media data box 21
8.2 Movie structure 21
8.2.1 Movie box 21
8.2.2 Movie header box 21
8.3 Track structure 23
8.3.1 Track box 23
8.3.2 Track header box 23
8.3.3 Track reference box 26
8.3.4 Track group box 28

8.3.5 Track type box 29
8.4 Track media structure 30
8.4.1 Media box 30
8.4.2 Media header box 30
8.4.3 Handler reference box 31
8.4.4 Media information box 32
8.4.5 Media information header boxes 32
8.4.6 Extended language tag 32
8.5 Sample tables 33
8.5.1 Sample table box 33
8.5.2 Sample description box 34
8.5.3 Degradation priority box 36
8.5.4 Sample scale box 36
8.6 Track time structures 36
8.6.1 Time to sample boxes 36
8.6.2 Sync sample box 41
8.6.3 Shadow sync 42
8.6.4 Independent and disposable samples box 43
8.6.5 Edit box 45
8.6.6 Edit list box 45
8.7 Track data layout structures 48
8.7.1 Data information box 48
8.7.2 Data reference box 48
8.7.3 Sample size boxes 50
8.7.4 Sample to chunk box 51
8.7.5 Chunk offset box 52
8.7.6 Padding bits box 52
8.7.7 Sub-sample information box 53
8.7.8 Sample auxiliary information sizes box 54
8.7.9 Sample auxiliary information offsets box 56
8.8 Movie fragments 57
8.8.1 Movie extends box 57
8.8.2 Movie extends header box 58
8.8.3 Track extends box 58
8.8.4 Movie fragment box 59
8.8.5 Movie fragment header box 60
8.8.6 Track fragment box 60
8.8.7 Track fragment header box 60
8.8.8 Track fragment run box 62
8.8.9 Movie fragment random access box 63
8.8.10 Track fragment random access box 64
8.8.11 Movie fragment random access offset box 65
8.8.12 Track fragment decode time box 65
8.8.13 Level assignment box 66
8.8.14 Sample auxiliary information in movie fragments 68
8.8.15 Track Extension Properties box 68
8.8.16 Alternative startup sequence properties box 68
8.8.17 Metadata and user data in movie fragments 69
8.9 Sample group structures 70
8.9.1 Overview 70
8.9.2 Sample to group box 70
8.9.3 Sample group description box 72
8.9.4 Representation of group structures in movie fragments 74
8.9.5 Compact sample to group box 75
8.10 User data 77
8.10.1 User data box 77
8.10.2 Copyright box 77
8.10.3 Track selection box 78

8.10.4 Track kind 79
8.11 Metadata support 80
8.11.1 MetaBox 80
8.11.2 XML boxes 81
8.11.3 Item location box 81
8.11.4 Primary item box 84
8.11.5 Item protection box 85
8.11.6 Item information box 85
8.11.7 Additional metadata container box 87
8.11.8 Metabox Relation box 87
8.11.9 URL forms for MetaBoxes 88
8.11.10 Static metadata 88
8.11.11 Item data box 89
8.11.12 Item reference box 89
8.11.13 Auxiliary video metadata 90
8.11.14 Item properties box 90
8.11.15 Brand item property 92
8.12 Support for protected streams 93
8.12.1 Overview 93
8.12.2 Protection scheme information box 94
8.12.3 Original format box 94
8.12.4 IPMPInfoBox 95
8.12.5 IPMP control box 95
8.12.6 Scheme type box 95
8.12.7 Scheme information box 95
8.12.8 Scramble Scheme Information Box 96
8.13 File delivery format support 96
8.13.1 Overview 96
8.13.2 FD item information box 97
8.13.3 File partition box 97
8.13.4 FEC reservoir box 99
8.13.5 FD session group box 99
8.13.6 Group ID to name box 100
8.13.7 File reservoir box 101
8.14 Sub tracks 101
8.14.1 Overview 101
8.14.2 Backward compatibility 102
8.14.3 Sub track box 102
8.14.4 Sub track information box 102
8.14.5 Sub track definition box 103
8.14.6 Sub track sample group box 104
8.15 Post-decoder requirements on media 104
8.15.1 General 104
8.15.2 Restricted sample entry transformation 105
8.15.3 Restricted scheme information box 105
8.15.4 Scheme for stereoscopic video arrangements 106
8.15.5 Compatible scheme type box 108
8.16 Segments 108
8.16.1 Overview 108
8.16.2 Segment type box 108
8.16.3 Segment index box 109
8.16.4 Subsegment index box 112
8.16.5 Producer reference time box 114
8.17 Support for incomplete tracks 115
8.17.1 General 115
8.17.2 Transformation 116
8.17.3 Complete track information box 116
8.18 Entity grouping 117

8.18.1 General ... 117
8.18.2 Groups list box ... 117
8.18.3 Entity to group box ... 117
8.19 Compressed boxes ... 118
8.19.1 Overview and processing ... 118
8.19.2 Processing model ... 119
8.19.3 General syntax ... 120
8.19.4 General semantics ... 120
8.19.5 Original file-type box ... 120
8.19.6 Compressed movie box ... 121
8.19.7 Compressed movie fragment box ... 121
8.19.8 Compressed segment index box ... 121
8.19.9 Compressed subsegment index box ... 122
9 Hint track formats ... 122
9.1 RTP and SRTP hint track format ... 122
9.1.1 Overview ... 122
9.1.2 Sample description format ... 123
9.1.3 Sample format ... 124
9.1.4 SDP information ... 127
9.1.5 Statistical information ... 127
9.2 ALC/LCT and FLUTE hint track format ... 128
9.2.1 Overview ... 128
9.2.2 Design principles ... 129
9.2.3 Sample description format ... 130
9.2.4 Sample format ... 130
9.3 MPEG-2 transport hint track format ... 133
9.3.1 Overview ... 133
9.3.2 Design principles ... 134
9.3.3 Sample description format ... 135
9.3.4 Sample format ... 137
9.3.5 Protected MPEG 2 transport stream hint track ... 139
9.4 RTP, RTCP, SRTP and SRTCP reception hint tracks ... 140
9.4.1 RTP reception hint track ... 140
9.4.2 RTCP reception hint track ... 143
9.4.3 SRTP reception hint track ... 144
9.4.4 SRTCP reception hint tracks ... 146
9.4.5 Protected RTP reception hint track ... 147
9.4.6 Recording procedure ... 147
9.4.7 Parsing procedure ... 147
10 Sample groups ... 147
10.1 Random access recovery points ... 147
10.1.1 Definition ... 147
10.1.2 Syntax ... 148
10.1.3 Semantics ... 148
10.2 Rate share groups ... 148
10.2.1 Overview ... 148
10.2.2 Rate share sample group entry ... 149
10.2.3 Relationship between tracks ... 150
10.2.4 Bitrate allocation ... 151
10.3 Alternative startup sequences ... 151
10.3.1 Definition ... 151
10.3.2 Syntax ... 152
10.3.3 Semantics ... 152
10.3.4 Examples ... 152
10.4 Random access point (RAP) sample group ... 154
10.4.1 Definition ... 154
10.4.2 Syntax ... 154

10.4.3 Semantics 154
10.5 Temporal level sample group 154
10.5.1 Definition 154
10.5.2 Syntax 155
10.5.3 Semantics 155
10.6 Stream access point sample group 155
10.6.1 Definition 155
10.6.2 Syntax 155
10.6.3 Semantics 155
10.7 Sample-to-item sample group 156
10.7.1 Definition 156
10.7.2 Syntax 156
10.7.3 Semantics 156
10.8 Dependent random access point (DRAP) sample group 156
10.8.1 Definition 156
10.8.2 Syntax 157
10.8.3 Semantics 157
10.9 Pixel Aspect Ratio Sample Grouping 157
10.9.1 Definition 157
10.9.2 Syntax 157
10.9.3 Semantics 157
10.10 Clean Aperture Sample Grouping 157
10.10.1 Definition 157
10.10.2 Syntax 158
10.10.3 Semantics 158

11 Derived file formats 158

12 Media-specific definitions 159
12.1 Video media 159
12.1.1 Media handler 159
12.1.2 Video media header 159
12.1.3 Sample entry 159
12.1.4 Pixel aspect ratio and clean aperture 160
12.1.5 Colour information 162
12.1.6 Content light level 163
12.1.7 Mastering display colour volume 163
12.1.8 Content colour volume 163
12.1.9 Ambient viewing environment 164
12.2 Audio media 164
12.2.1 Media handler 164
12.2.2 Sound media header 164
12.2.3 Sample entry 165
12.2.4 Channel layout 167
12.2.5 Downmix instructions 169
12.2.6 DRC information 172
12.2.7 Audio stream loudness 173
12.3 Metadata media 175
12.3.1 Media handler 175
12.3.2 Media header 175
12.3.3 Sample entry 175
12.4 Hint media 177
12.4.1 Overview 177
12.4.2 Media handler 178
12.4.3 Hint media header 178
12.4.4 Sample entry 178
12.5 Text media 179
12.5.1 Media handler 179
12.5.2 Media header 179

12.5.3 Sample entry 179
12.6 Subtitle media 179
12.6.1 Media handler 179
12.6.2 Subtitle media header 179
12.6.3 Sample entry 180
12.7 Font media 181
12.7.1 Media handler 181
12.7.2 Media header 181
12.7.3 Sample entry 181
12.8 Transformed media 181
12.8.1 General 181
12.8.2 Multiple transformations for a single transformed media track 181
12.8.3 Determining the untransformed sample entry type 182
12.8.4 The `'codecs'` MIME parameter for a transformed media track 182
12.9 Multiplexed timed metadata tracks 182
12.9.1 General 182
12.9.2 Overall design 182
12.9.3 Sample format 183
12.9.4 Sample entry format 183
12.9.5 Defined formats 186
12.10 Volumetric visual media 187
12.10.1 Media handler 187
12.10.2 Media header 187
12.10.3 Sample entry 187
12.10.4 Sample format 188
12.11 Haptic media 188
12.11.1 Media handler 188
12.11.2 Media header 188
12.11.3 Sample entry 188
12.11.4 Sample format 188

Annex A (informative) **Background and tutorial** **189**

Annex B (informative) **Guidance on deriving from this document** **197**

Annex C (normative) **Fragment identifiers for ISO base media resources** **206**

Annex D (informative) **Management of extension code-points** **207**

Annex E (normative) **File format brands** **209**

Annex F (normative) **MIME type registration of segments** **220**

Annex G (informative) **URI-labelled metadata forms** **221**

Annex H (informative) **Processing of RTP streams and reception hint tracks** **223**

Annex I (normative) **Stream access points** **240**

Annex J (informative) **Segment index examples** **243**

Annex K (normative) **Use of IETF RFC 6381 for ISOBMFF files** **246**

Bibliography **249**

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

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. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

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

This seventh edition cancels and replaces the sixth edition (ISO/IEC 14496-12:2020), which has been technically revised.

The main changes care as follows:

— re-organization of all the introductory material, such that it the material needed by a specific audience or applying to specific aspects is brought together;

— terminology with respect to timing is more consistent and simpler;

— replacing the word 'metadata' as describing the structural data, as the word metadata is also used with another meaning, and the dual use was confusing;

— providing better wording for the `TrackHeader` flags;

— other minor editorial improvements.

A list of all parts in the ISO/IEC 14496 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The ISO base media file format is designed to contain timed media information for a presentation in a flexible, extensible format that facilitates interchange, management, editing, and presentation of the media. This presentation may be 'local' to the system containing the presentation, or may be via a network or other stream delivery mechanism.

The file structure is object-oriented; a file can be decomposed into constituent objects very simply, and the structure of the objects inferred directly from their type.

The file format is designed to be independent of any particular network protocol while enabling efficient support for them in general.

The ISO base media file format is a base format for media file formats.

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

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights

The holders of these patent rights have assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

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

Information technology — Coding of audio-visual objects —

Part 12: ISO base media file format

1 Scope

This document specifies the ISO base media file format, which is a general format forming the basis for a number of other more specific file formats. This format contains the timing, structure, and media information for timed sequences of media data, such as audio-visual presentations.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ITU-T X.667/ ISO/IEC 9834-8, *Information technology – Procedures for the operation of object identifier registration authorities – Part 8: Generation of universally unique identifiers (UUIDs) and their use in object identifiers*

ISO/IEC 10646, *Information technology — Universal coded character set (UCS)*

ISO/IEC 13818-2:2013, *Information technology — Generic coding of moving pictures and associated audio information — Part 2: Video*

ISO/IEC 14496-1, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-10:2014, *Information technology — Coding of audio-visual objects — Part 10: Advanced Video Coding*

ISO 15076-1, *Image technology colour management — Architecture, profile format and data structure — Part 1: Based on ICC.1:2010*

ISO/IEC 15938-1, *Information technology — Multimedia content description interface — Part 1: Systems*

ISO/IEC 23001-1, *Information technology — MPEG systems technologies — Part 1: Binary MPEG format for XML*

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

ISO/IEC 23003-4, *Information technology — MPEG audio technologies — Part 4: Dynamic range control*

ITU-T H.265 / ISO/IEC 23008-2, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 2: High efficiency video coding*

ISO/IEC 23091-2, *Information technology —Coding-independent code points — Part 2: Video*

ISO/IEC 23091-3, *Information technology — Coding-independent code points — Part 3: Audio*

IETF RFC 1951, *DEFLATE Compressed Data Format Specification version 1.3*

IETF RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*

IETF RFC 2046, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*

IETF RFC 3629, *UTF-8, a transformation format of ISO 10646*

IETF RFC 3711:2004, *The Secure Real-time Transport Protocol (SRTP)*

IETF RFC 5052, *Forward Error Correction (FEC) Building Block*

IETF RFC 5905, *Network Time Protocol Version 4: Protocol and Algorithms Specification*

ITU-R TF.460-6:2002, *Standard-frequency and time-signal emissions*

ITU-R BS.1770-4, *Algorithms to measure audio programme loudness and true-peak audio level*

IETF BCP 47, *Tags for Identifying Languages*

IETF RFC 4122, *A Universally Unique IDentifier (UUID) URN Namespace*

IETF RFC 3061, *A URN Namespace of Object Identifiers*

W3C Recommendation, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 26 November 2008, https://www.w3.org/TR/2008/REC-xml-20081126/

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

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

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1.1
box
object-oriented building block defined by a unique type identifier and length

Note 1 to entry: Called ‘atom’ in some specifications, including the first definition of MP4

3.1.2
chunk
contiguous set of samples for one track

3.1.3
clean aperture
part of a decoded video image from which undesirable pixels introduced for coding purposes such as having integer number of coding blocks have been removed for presentation

3.1.4
container box
box whose sole purpose is to contain and group a set of related boxes

Note 1 to entry: Container boxes are normally not derived from `FullBox`

3.1.5
file level
byte position in an ISO base media file not contained in a Box structure

3.1.6
full aperture
decoded video image as output by the decoder which may contain undesirable pixels for presentation

3.1.7
hint track
special track which does not contain media data, but instead contains instructions for packaging one or more tracks into a streaming channel

3.1.8
hinter
tool that is run on a file containing only media, to add one or more hint tracks to the file and so facilitate streaming

3.1.9
index file
ISO base media file containing only `SegmentIndexBox`

3.1.10
ISO base media file
file conforming to the file format described in this document (either a movie file, a metadata file, a segment file or an index file)

3.1.11
item
data which does not require timed processing, as opposed to sample data

3.1.12
leading sample
sample associated with a random access point (RAP) that precedes the RAP in composition order and immediately follows the RAP or another leading sample in decoding order, and which possibly cannot be correctly decoded when decoding starts from the RAP

3.1.13
leaf subsegment
subsegment that does not contain any indexing information that would enable its further division into subsegments

3.1.14
mod
modulo operator: (x mod y) = x-y floor (x/y)

3.1.15
media data box
box which can hold the actual media data for a presentation (`'mdat'`)

3.1.16
metadata file
ISO base media file containing a top-level `MetaBox`

Note 1 to entry: A Movie File may also be a Metadata File, and vice-versa.

3.1.17
movie box
container box whose sub-boxes define the structure-data for a presentation (`'moov'`)

3.1.18
movie file
ISO base media file containing a `MovieBox`

3.1.19
movie fragment
fragment of the information contained in a `MovieBox`, defined by a `MovieFragmentBox` and its contents

3.1.20
structure-data
data that provides the location, size, timing, and characteristics of the media data (e.g. the coded frames of audio and video)

3.1.21
movie-fragment relative addressing
signalling of offsets for media data in movie fragments that is relative to the start of those movie fragments, specifically setting the flags base-data-offset-present to 0 and default-base-is-moof to 1 in `TrackFragmentHeaderBox`es

Note 1 to entry: Setting the default-base-is-moof flag to 1 is only relevant for movie fragments that contain more than one track run (either in the same or several tracks).

3.1.22
open random access point
sample after which all samples in composition order can be correctly decoded, but some samples following the random access point in decoding order and preceding the random access point in composition order need not be correctly decodable

Note 1 to entry: For example, an intra picture starting an open group of pictures can be followed in decoding order by (bi-)predicted pictures that however precede the intra picture in composition order; though they possibly cannot be correctly decoded if the decoding starts from the intra picture, they are not needed.

3.1.23
pixel aspect ratio
scaling required to be applied to the output pixel of a decoder to produce a non-distorted image

Note 1 to entry: The term "Sample Aspect Ratio" is sometimes used for this term, but "sample" in this standard has a specific meaning.

3.1.24
presentation
one or more motion sequences, possibly combined with audio

3.1.25
presentation time
timeline of a track that aligns with the timelines of other tracks established by an explicit or implied edit list applied to the composition timestamps

3.1.26
timeline
monotonic linear representation of times with respect to a zero origin point

3.1.27
timescale
number of timestamp values that represent a duration of one second

3.1.28
timestamp
integer coded value representing an instant of time on an associated timeline

3.1.29
decoding timestamp
timestamp on the media timeline such that samples are decoded in decoding timestamp order

Note 1 to entry: The decoding timestamps primarily define the required decoding order; systems may decode at a time of their choosing; in the case of signed composition offsets, decoding timestamps do not necessarily precede composition timestamps.

3.1.30
composition timestamp
timestamp on the media timeline such that samples are presented in composition timestamp order and that establishes their relative composition timing

3.1.31
sample duration
difference between the decoding timestamp of the following sample (when known) and the decoding timestamp of this sample

3.1.32
random access point
RAP
sample in a track that starts at the ISAU of a SAP of type 1 or 2 or 3; informally, a sample, from which when decoding starts, the sample itself and all samples following in composition order can be correctly decoded

Note 1 to entry: SAP types are defined in Annex I.

3.1.33
random access recovery point
sample in a track with presentation time equal to the TSAP of a SAP of type 4; informally, a sample, that can be correctly decoded after having decoded a number of samples that is before this sample in decoding order, sometimes known as gradual decoding refresh

Note 1 to entry: SAP types are defined in Annex I.

3.1.34
sample
all the data associated with a single time

Note 1 to entry: No two samples within a track can share the same decoding timestamp; no two samples can share the same composition timestamp.

Note 2 to entry: In non-hint tracks, a sample is, for example, an individual frame of video, a series of video frames in decoding order, or a compressed section of audio in decoding order; in hint tracks, a sample defines the formation of one or more streaming packets.

3.1.35
sample description
structure which defines and describes the format of some number of samples in a track

3.1.36
sample entry type
four-character code that is either a format value of a `SampleEntry` directly contained in `SampleDescriptionBox` or a `data_format` value of `OriginalFormatBox`

3.1.37
untransformed sample entry type
sample entry type of the track that would apply if no transformations had been performed to a transformed media track

Note 1 to entry: This is the *sample entry type* that would be the `format` value in a `SampleEntry` directly contained in the `SampleDescriptionBox`.

3.1.38
sample number
ordinal index number of a given sample where the first sample has sample number 1

3.1.39
sample table
packed directory for the timing and physical layout of the samples in a track

3.1.40
sync sample
sample in a track that starts at the ISAU of a SAP of type 1 or 2

Note 1 to entry: SAP types are defined in Annex I.

Note 2 to entry: Informally, a media sample that starts a new independent sequence of samples; if decoding starts at the sync sample, it and succeeding samples in decoding order can all be correctly decoded, and the resulting set of decoded samples forms the correct presentation of the media starting at the decoded sample that has the earliest composition time; a media format may provide a more precise definition of a sync sample for that format.

3.1.41
segment
portion of movie file, consisting of either (a) a `MovieBox`, with its associated media data (if any) and other associated boxes or (b) one or more `MovieFragmentBox`es, with their associated media data, and other associated boxes

Note 1 to entry: The associated media data can be found by following byte offsets, but the process of finding associated boxes is not given in this standard and may be derived by other specifications.

3.1.42
segment file
ISO base media file containing one or more segment(s)

3.1.43
subsegment
time interval of a segment formed from `MovieFragmentBox`es, that is also a valid segment

3.1.44
thumbnail image
smaller-resolution representation of an image

3.1.45
top-level box
box contained at file level

3.1.46
track
timed sequence of related samples (q.v.) in an ISO base media file

Note 1 to entry: For media data, a track corresponds to a sequence of images or sampled audio; for hint tracks, a track corresponds to a streaming channel.

3.1.47
haptic media
timed tactile signals to be presented as part of the media presentation

3.1.48
volumetric visual media
timed visual media defining a visual coding in a three-dimensional space

Note 1 to entry: In contrast to video media, which defines a planar coding.

3.2 Abbreviated terms

ALC	asynchronous layered coding
AVC	advanced video coding
FD	file delivery
FDT	file delivery table
FEC	forward error correction
FLUTE	file delivery over unidirectional transport
IANA	internet assigned numbers authority
IETF	internet engineering task force
LCT	layered coding transport
MBMS	multimedia broadcast/multicast service
MIME	multipurpose internet mail extensions (as defined in IETF RFC 2045 and IETF RFC 2046)
MVC	multiview video coding
SVC	scalable video coding
UUID	universally unique identifier (as defined in IETF RFC 4122 and ISO/IEC 9834-8)

4 Object-structured file organization

4.1 File structure

Files are formed as a series of objects, called boxes in this document. All data is contained in boxes; there is no other data within the file. This includes any initial signature required by the specific file format.

All object-structured files conformant to Clause 4 (all object-structured files) shall contain a `FileTypeBox`.

In this document, top-level boxes (boxes not contained in other boxes) are indicated as being at 'file' level, with the notation "Container: File".

4.2 Object structure

4.2.1 Object syntax conventions

The definitions of objects are given in the syntax description language (SDL) defined in ISO/IEC 14496-1.

NOTE Comments in the code fragments in this document are informative.

The fields in the objects are stored with the most significant byte first, commonly known as network byte order or big-endian format. When fields smaller than a byte are defined, or fields span a byte boundary, the bits are assigned from the most significant bits in each byte to the least significant. For example, a field of two bits followed by a field of six bits has the two bits in the high order bits of the byte.

In the SDL, the notation '=' indicates that in this object, the field read from the bitstream (whose name and type are given in the left hand-side part) is expected to match the value given in the right hand-side part.

The following basic field types are defined. In these definitions, null-terminated means that the last character of a string is Unicode NUL, and hence an empty string is represented by a single Unicode NUL. Some fields using these types may restrict the characters permitted.

Name	Semantics
`utf8string`	UTF-8 string as defined in IETF RFC 3629, null-terminated.
`utfstring`	null-terminated string encoded using either UTF-8 or UTF-16. If UTF-16 is used, the sequence of bytes shall start with a byte order mark (BOM) and the null termination shall be 2 bytes set to 0.
`utf8list`	null-terminated list of space-separated UTF-8 strings
`base64string`	null-terminated base64 encoded data

4.2.2 Object definitions

An object in this terminology is a box.

Boxes start with a header which gives both size and type. The header permits compact or extended size (32 or 64 bits) and compact or extended types (32 bits or full universal unique identifiers, i.e. UUIDs). The standard boxes all use compact types (32-bit) and most boxes will use the compact (32-bit) size. Typically, only the `MediaDataBox` needs the 64-bit size.

To permit ease of identification, the 32-bit compact type can be expressed as four characters from the range 0020 to 007E, inclusive, of ISO/IEC 10646 (technically identical to the Unicode standard[28]) or ISO/IEC 8859-1[34]. Each character is hence expressible in a single byte. The four individual byte values of the field are placed in order in the file. Other fields may also use this 32-bit representation, referred to as a 'four-character code' (4CC). The maintenance of four-character codes used in the format is defined in Annex D.

The size is the entire size of the box, including the size and type header, fields, and all contained boxes. This facilitates general parsing of the file.

User extensions use an extended box type; in this case, the type field is set to `'uuid'`.

```
aligned(8) class BoxHeader (
      unsigned int(32) boxtype,
      optional unsigned int(8)[16] extended_type) {
   unsigned int(32) size;
   unsigned int(32) type = boxtype;
   if (size==1) {
      unsigned int(64) largesize;
   } else if (size==0) {
      // box extends to end of file
   }
   if (boxtype=='uuid') {
      unsigned int(8)[16] usertype = extended_type;
   }
}
aligned(8) class Box (
      unsigned int(32) boxtype,
      optional unsigned int(8)[16] extended_type) {
   BoxHeader(boxtype, extended_type);
   // the remaining bytes are the BoxPayload
}
```

The semantics of these two fields are:

`size` is an integer that specifies the number of bytes in this box, including all its fields and contained boxes; if size is 1 then the actual size is in the field `largesize`; if size is 0, then this box shall be in a top-level container, and be the last box in that container (typically, a file or data object delivered over a protocol), and its payload extends to the end of that container (normally only used for a MediaDataBox)

type identifies the box type; user extensions use an extended type, and in this case, the type field is set to 'uuid'.

BoxPayload is defined as all the bytes in a Box, included by the size or largesize field (as appropriate), following the BoxHeader.

Boxes with an unrecognized type shall be ignored and skipped.

Many objects also contain a version number and flags field:

```
aligned(8) class FullBoxHeader(unsigned int(8) v, bit(24) f)
{
   unsigned int(8)   version = v;
   bit(24)            flags = f;
}
aligned(8) class FullBox(unsigned int(32) boxtype,
      unsigned int(8) v, bit(24) f,
      optional unsigned int(8)[16] extended_type)
   extends Box(boxtype, extended_type)
{
   FullBoxHeader(v, f);
   // the remaining bytes are the FullBoxPayload
}
```

The semantics of these two fields are:

version is an integer that specifies the version of this format of the box.

flags is a map of flags

FullBoxPayload is defined as all the bytes in a FullBox, included by the size or largesize field (as appropriate), following the FullBoxHeader. Since BoxPayload is defined for any box extending from Box, FullBox has both BoxPayload and FullBoxPayload defined, the former including the latter (with BoxPayload of a FullBox including the FullBoxHeader).

The payload (contents) of a box are FullBoxPayload for a FullBox and BoxPayload for all boxes.

FullBoxes with an unrecognized version shall be ignored and skipped.

NOTE This document and derived specifications describe the different flags defined for a box extending from FullBox as hexadecimal values (e.g. 0x000001); these values correspond to 24-bit unsigned integers with the most significant byte first.

4.2.3 Extensibility of object definitions

The normative objects defined in this document are identified by a 32-bit value, which is normally a four character compact code as defined in 4.2.

To permit user extension of the format, to store new object types, and to permit the inter-operation of the files formatted to this documentwith certain distributed computing environments, there are a type mapping and a type extension mechanism that together form a pair.

Commonly used in distributed computing are UUIDs (universal unique identifiers), which are 16 bytes. Any normative type specified here can be mapped directly into the UUID space by composing the four byte type value with the twelve byte ISO reserved value, 0xXXXXXXXX-0011-0010-8000-00AA00389B71. The four character code replaces the XXXXXXXX in the preceding number. These types are identified to ISO as the object types used in this document.

User objects use the escape type 'uuid'. They are documented above in subclause 6.3. After the size and type fields, there is a full 16-byte UUID.

Systems which wish to treat every object as having a UUID could employ the following algorithm:

```
size := read_uint32();
type := read_uint32();
if (type=='uuid')
   then uuid := read_uuid()
   else uuid := form_uuid(type, ISO_12_bytes);
```

Similarly when linearizing a set of objects into files formatted to this document, the following is applied:

```
write_uint32( object_size(object) );
uuid := object_uuid_type(object);
if (is_ISO_uuid(uuid) )
   write_uint32( ISO_type_of(uuid) )
   else { write_uint32('uuid'); write_uuid(uuid); }
```

A file containing boxes from this document that have been written using the `'uuid'` escape and the full UUID is not compliant; systems are not required to recognize standard boxes written using the `'uuid'` and an ISO UUID.

4.3 File-type box

4.3.1 Definition

Box Type: `'ftyp'`
Container: File, or `OriginalFileTypeBox`
Mandatory: Yes
Quantity: Exactly one (but see below)

Files conformant to Clause 4 shall contain a `FileTypeBox`. For compatibility with an earlier edition of this document, files may be conformant to this document and not contain a `FileTypeBox`. Files with no `FileTypeBox` should be read as if they contained a `FileTypeBox` with `Major_brand='mp41', minor_version=0`, and the single compatible brand `'mp41'`.

A media-file structured to this document may be compatible with more than one detailed specification, and it is therefore not always possible to speak of a single 'type' or 'brand' for the file. This means that the utility of the file name extension and multipurpose internet mail extension (MIME) type are somewhat reduced.

This box shall be placed as early as possible in the file (e.g. after any obligatory signature, but before any significant variable-size boxes such as a `MovieBox`, `MediaDataBox`, or `FreeSpaceBox`). It identifies which specification is the 'best use' of the file, and a minor version of that specification; and also a set of other specifications to which the file complies. Readers implementing this format should attempt to read files that are marked as compatible with any of the specifications that the reader implements. Any incompatible change in a specification should therefore register a new 'brand' identifier to identify files conformant to the new specification.

The minor version is informative only. It does not appear for compatible-brands, and is not used to determine the conformance of a file to a standard. It may allow more precise identification of the major specification, for inspection, debugging, or improved decoding.

Files would normally be externally identified (e.g. with a file extension or mime type) that identifies the 'best use' (major brand), or the brand that the author believes will provide the greatest compatibility.

This subclause does not define any brands. However, see subclause 6.3 for brands for files conformant to the whole document and not just this subclause. All file format brands defined in this document are included in Annex E with a summary of which features they require.

4.3.2 Syntax

```
aligned(8) class GeneralTypeBox(code) extends Box(code) {
   unsigned int(32)   major_brand;
   unsigned int(32)   minor_version;
   unsigned int(32)   compatible_brands[];   // to end of the box
}
```

```
aligned(8) class FileTypeBox extends GeneralTypeBox ('ftyp')
{}
```

4.3.3 Semantics

This box identifies the specifications to which this file complies.

Each brand is a four character code, registered with ISO, that identifies a precise specification.

`major_brand` – is a brand identifier

`minor_version` – is an informative integer for the minor version of the major brand

`compatible_brands` – is a list, to the end of the box, of brands

4.4 Extended type box

4.4.1 Definition

Box Type: `'etyp'`
Container: File, `ItemPropertyContainerBox`, or `OriginalFileTypeBox`
Mandatory: No
Quantity: Zero or more in file, zero or one per an item

Box Type: `'tyco'`
Container: `ExtendedTypeBox`
Mandatory: Yes
Quantity: One or more

The `ExtendedTypeBox` may be placed after the `FileTypeBox`, any `SegmentTypeBox`, or any `TrackTypeBox`, or used as an item property to indicate that a reader should only process the file, the segment, the track, or the item, respectively, if it supports the processing requirements of all the brands in at least one of the contained `TypeCombinationBox`es, or at least one brand in the preceding `FileTypeBox`, `SegmentTypeBox`, or `TrackTypeBox`, or in the `BrandProperty` associated with the same item, respectively.

The `TypeCombinationBox` expresses that the associated file, segment, track, or item may contain any boxes or other code points required to be supported in any of the brands listed in the `TypeCombinationBox` and that the associated file, segment, track, or item complies with the intersection of the constraints of the brands listed in the `TypeCombinationBox`.

NOTE Effectively, if the `compatible_brands` in the `FileTypeBox` are labeled `C[1], C[2]`, etc., and the `compatible_brands` in the first `TypeCombinationBox D[1,1], D[1,2]`, and the second `D[2,1], D[2,2]`, and so on, then a reader has to support:

```
C[1]
or C[2]
or …
or ( D[1,1] and D[1,2] and …)
or ( D[2,1] and D[2,2] and …)
…
```

4.4.2 Syntax

```
aligned(8) class TypeCombinationBox extends Box('tyco') {
   unsigned int(32)   compatible_brands[];   // to end of the box
}
aligned(8) class ExtendedTypeBox extends Box('etyp') {
   TypeCombinationBox   compatible_combinations[];   // to end of the box
}
```

4.4.3 Semantics

```
compatible_brands is a list of brands.
compatible_combinations is a list of TypeCombinationBoxes.
```

5 Structure of this document

Clause 4 defines object-structured files; files that are built from boxes.

Clause 6 gives the core concepts and data-types for time-based presentations, called 'movies' in this document.

Clause 8 defines the boxes used by time-based presentations, and other formats.

Clause 9 defines the hint track formats used to support some streaming protocols.

Clause 10 builds on the concept of sample groups as defined in 8.9 and defines some sample groups.

Clause 11 defines how to base a file format on this specification.

Clause 12 builds on the general concepts of tracks as defined in Clause 8, and defines track formats for various general types of media (video, sound, etc.).

Annex A provides an informative introduction to time-based presentations, which may be of assistance to first-time readers and implementers.

Annex B provides guidance on writing derived specifications.

Annex C provides the syntax for uniform resource identifier (URI) fragments.

Annex D documents how identifier values defined externally to this document are managed.

Annex E defines brands that may be used to identify the conformance and reader requirements to the structures defined in this document for time-based presentations.

Annex F contains the formal IANA registration of segments.

Annex G defines some forms used for labelling metadata with uniform resource identifier (URI) labels.

Annex H provides an overview of the use of hint tracks for RTP streams and RTP stream reception.

Annex I contains the formal definitions of the types of stream access points in timed media streams.

Annex J contains examples of the use of the `SegmentIndexBox` defined in 8.16.3.

Annex K defines the MIME parameters that may be used to annotate MIME types for time-based presentations based on Clause 6.

6 ISO base media file organization

6.1 Files, segments, and streams

This document supports the exchange of presentations in three principle ways:

1) As a single file (e.g. on exchangeable media such as discs, or as a download).

2) As a series of segments, preceded by an initialization segment.

3) Transformed by supporting structures, called hint tracks, into a streaming protocol such as the IETF real-time protocol RTP [[ref]] or an MPEG-2 transports stream [[ref]].

A presentation file logically includes all its segments.

The file format supports transformation of media data into a streaming protocol 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.

6.2 Presentation structure

6.2.1 Object structure of a presentation

The presentation file is an object-structured file as defined in Clause 4 (and hence contains a `FileTypeBox`); some of these objects may contain other objects.

Presentation files contain data that structures, orders, times, and describes the media data that is passed to decoders. This non-media data is called structure-data here.

The sequence of objects in a presentation shall contain exactly one `MovieBox`. It is usually close to the beginning or end of the file, to permit its easy location. The other objects found at this level include a `FileTypeBox`, `FreeSpaceBox(es)`, `MovieFragmentBox(es)`, a `MetaBox`, or `MediaDataBox`(es).

6.2.2 Meta data and media data

The structure-data is contained within a structure-data wrapper (the `MovieBox` or `MovieFragmentBox`); the media data is contained either in the same file, within `MediaDataBox`(es), or in other files. The media data is composed, for example, of images or audio data; the media data containers, or media data files, may contain other un-referenced information.

The file containing the presentation structure-data may also contain all the media data, whereupon the presentation is self-contained. If the media data is externally referenced in other files, they are not required to be formatted to this document; they are used to contain media data, and may also contain unused media data, or other information. This document concerns the structure of the presentation file only. The format of the media-data files is constrained by this document only in that the media-data in the media files needs to be capable of description by the structure-data defined here.

These other files may be ISO files, image files, or other formats. Only the media data itself, such as JPEG 2000 images, is stored in these other files; all timing and framing (position and size) information is in the ISO base media file, so the ancillary files are essentially free-format.

6.3 Structure-data (objects)

6.3.1 Box

Type fields not defined here are reserved. Private extensions shall be achieved through the `'uuid'` type. In addition, the following types are not and will not be used, or used only in their existing sense, in future editions of this document, to avoid conflict with existing content using earlier pre-standard versions of this format:

`clip, crgn, matt, kmat, pnot, ctab, load, imap;`

these track reference types (as found in the `reference_type` of a `TrackReferenceBox`):

`tmcd, chap, sync, scpt, ssrc.`

A number of boxes contain index values into sequences in other boxes. These indexes start with the value 1 (1 is the first entry in the sequence).

6.3.2 Data types and fields

In a number of boxes in this document, there are two variant forms: version 0 using 32-bit fields, and version 1 using 64-bit sizes for those same fields. In general, if a version 0 box (32-bit field sizes) can be used, it should be; version 1 boxes should be used only when the 64-bit field sizes they permit, are required. Values for counters, offsets, times, durations etc. in this format do not ‘wrap’ to 0 when the maximum value that can be stored in their field is reached; appropriately large fields must be used for all values.

For convenience during content creation there are creation and modification times stored in the file. These can be 32-bit or 64-bit numbers, counting seconds since midnight, Jan. 1, 1904, which is a convenient date for leap-year calculations. 32 bits are sufficient until approximately year 2040. These times shall be expressed in Universal Time Coordinated (UTC) as defined in ITU-R TF.460-6:2002, Annex I, and therefore may need adjustment to local time if displayed.

Fixed-point numbers are signed or unsigned values resulting from dividing an integer by an appropriate power of 2. For example, a 30.2 fixed-point number is formed by dividing a 32-bit integer by 4.

Fields shown as "`template`" in the box descriptions are fields which are coded with a default value unless a derived specification defines their use and permits writers to use other values than the default. If the field is used in another specification, that use must be conformant with its definition here, and the specification must define whether the use is optional or mandatory. Similarly, fields marked "pre-defined" were used in an earlier edition of this document. For both kinds of fields, if a field of that kind is not used in a specification, then it should be set to the indicated default value. If the field is not used it shall be copied un-inspected when boxes are copied, and ignored on reading.

Matrix values which occur in the headers specify a transformation of video images for presentation. Not all derived specifications use matrices; if they are not used, they shall be set to the identity matrix. If a matrix is used, the point (p,q) is transformed into (p', q') using the matrix as follows:

```
(p q 1) *    | a   b   u   |    = (m n z)
             | c   d   v   |
             | x   y   w   |

m = ap + cq + x;   n = bp + dq + y;   z = up + vq + w;

p' = m/z;   q' = n/z
```

The coordinates {p,q} are on the decompressed frame, and {p', q'} are at the rendering output. Therefore, for example, the matrix {2,0,0, 0,2,0, 0,0,1} exactly doubles the pixel dimension of an image. The coordinates transformed by the matrix are not normalized in any way, and represent actual sample locations. Therefore {x,y} can, for example, be considered a translation vector for the image.

The coordinate origin is located at the upper left corner, and X values increase to the right, and Y values increase downwards. {p,q} and {p',q'} are to be taken as absolute pixel locations relative to the upper left hand corner of the original image (after scaling to the size determined by the track header's width and height) and the transformed (rendering) surface, respectively.

Each track is composed using its matrix as specified into an overall image; this is then transformed and composed according to the matrix at the movie level in the `MovieHeaderBox`. It is application-dependent whether the resulting image is 'clipped' to eliminate pixels, which have no display, to a vertical rectangular region within a window, for example. So for example, if only one video track is displayed and it has a translation to {20,30}, and a unity matrix is in the `MovieHeaderBox`, an application may choose not to display the empty "L" shaped region between the image and the origin.

All the values in a matrix are stored as 16.16 fixed-point values, except for u, v and w, which are stored as 2.30 fixed-point values.

The values in the matrix are stored in the order {a,b,u, c,d,v, x,y,w}.

6.3.3 URIs as type indicators

When URIs are used as a type indicator (e.g. in a sample entry or for un-timed metadata), the URI shall 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 shall 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).

6.3.4 Box order

An overall view of the normal encapsulation structure is provided in the following informative Table 1. In the event of a conflict between this table and the prose, the prose prevails. The order of boxes within its container is not necessarily indicated in Table 1.

The table shows those boxes that may occur at the top-level in the left-most column; indentation is used to show possible containment. Thus, for example, a `TrackHeaderBox` (tkhd) is found in a `TrackBox` (trak), which is found in a `MovieBox` (moov). Not all boxes need to be used in all files; the mandatory boxes are marked with an asterisk (*). See the description of the individual boxes for a discussion of what must be assumed if the optional boxes are not present.

Objects using an extended type may be placed in a wide variety of containers, not just the top level.

In order to improve interoperability and utility of the files, the following rules and guidelines shall be followed for the order of boxes:

1) The `FileTypeBox` shall occur before any variable-length box (e.g. movie, free space, media data). Only a fixed-size box such as a file signature, if required, may precede it.

2) It is strongly **recommended** that all header boxes be placed first in their container: these boxes are the `MovieHeaderBox`, `TrackHeaderBox`, `MediaHeaderBox`, and the specific media headers inside the `MediaInformationBox` (e.g. the `VideoMediaHeaderBox`).

3) Any movie fragment boxes **should** be in sequence order (see subclause 8.8.5).

4) It is **recommended** that the boxes within the `SampleTableBox` be in the following order: `SampleDescriptionBox`, `TimeToSampleBox`, `SampleToChunkBox`, `SampleSizeBox`, `ChunkOffsetBox`.

5) It is strongly **recommended** that the `TrackReferenceBox` and `EditBox` (if any) **should** precede the `MediaBox`, and the `HandlerBox` **should** precede the `MediaInformationBox`, and the `DataInformationBox` **should** precede the `SampleTableBox`.

6) It is **recommended** that `UserDataBox`es be placed last in their container.

7) It is **recommended** that the `MovieFragmentRandomAccessBox`, if present, be last in the file.

8) It is **recommended** that the `ProgressiveDownloadInfoBox` be placed as early as possible in files, for maximum utility.

Table 1 — Box types, structure and cross-reference

Box types, structure, and cross-reference								
ftyp						*	4.2.3	*file type and compatibility*
otyp							8.19.5	*original file-type*
pdin							8.1.3	*progressive download information*
moov						*	8.2.1	*container for all the structure-data*
	mvhd					*	8.2.2	*movie header, overall declarations*
	meta						8.11.1	*metadata*
	trak					*	8.3.1	*container for an individual track or stream*

Table 1 *(continued)*

Box types, structure, and cross-reference								
		tkhd				*	8.3.2	*track header, overall information about the track*
		tref					8.3.3	*track reference container*
		trgr					8.3.4	*track grouping indication*
		edts					8.6.5	*edit list container*
			elst				8.6.6	*an edit list*
		meta					8.11.1	*metadata*
		mdia				*	8.4	*container for the media information in a track*
			mdhd			*	8.4.2	*media header, overall information about the media*
			hdlr			*	8.4.3	*handler, declares the media (handler) type*
			elng				8.4.6	*extended language tag*
			minf			*	8.4.4	*media information container*
				vmhd			12.1.2	*video media header, overall information (video track only)*
				smhd			12.2.2	*sound media header, overall information (sound track only)*
				hmhd			12.4.3	*hint media header, overall information (hint track only)*
				sthd			12.6.2	*subtitle media header, overall information (subtitle track only)*
				nmhd			8.4.5.2	*Null media header, overall information (some tracks only)*
				dinf		*	8.7.1	*data information box, container*
					dref	*	8.7.2	*data reference box, declares source(s) of media data in track*
				stbl		*	8.5.1	*sample table box, container for the time/space map*
					stsd	*	8.5.2	*sample descriptions (codec types, initialization etc.)*
					stts	*	8.6.1.2	*(decoding) time-to-sample*
					ctts		8.6.1.3	*(composition) time to sample*
					cslg		8.6.1.4	*composition to decode timeline mapping*
					stsc	*	8.7.4	*sample-to-chunk, partial data-offset information*
					stsz		8.7.3.2	*sample sizes (framing)*
					stz2		8.7.3.3	*compact sample sizes (framing)*
					stco	*	8.7.5	*chunk offset, partial data-offset information*
					co64		8.7.5	*64-bit chunk offset*
					stss		8.6.2	*sync sample table*
					stsh		8.6.3	*shadow sync sample table*
					padb		8.7.6	*sample padding bits*
					stdp		8.7.6	*sample degradation priority*
					sdtp		8.6.4	*independent and disposable samples*
					sbgp		8.9.2	*sample-to-group*
					sgpd		8.9.3	*sample group description*
					subs		8.7.7	*sub-sample information*
					saiz		8.7.8	*sample auxiliary information sizes*
					saio		8.7.9	*sample auxiliary information offsets*
		udta					8.10.1	*user-data*
			cprt				8.10.2	*copyright etc.*
			tsel				8.10.3	*track selection box*
			kind				8.10.4	*track kind box*

Table 1 *(continued)*

Box types, structure, and cross-reference								
			strk				8.14.3	*sub track box*
				stri			8.14.4	*sub track information box*
				strd			8.14.5	*sub track definition box*
			ludt				12.2.7	*audio stream loudness*
	mvex						8.8.1	*movie extends box*
		mehd					8.8.2	*movie extends header box*
		trex				*	8.8.3	*track extends defaults*
		leva					8.8.13	*level assignment*
	udta						8.10.1	*user-data*
		cprt					8.10.2	*copyright etc.*
moof							8.8.4	*movie fragment*
	mfhd					*	8.8.5	*movie fragment header*
	meta						8.11.1	*metadata*
	traf						8.8.6	*track fragment*
		tfhd				*	8.8.7	*track fragment header*
		trun					8.8.8	*track fragment run*
		sbgp					8.9.2	*sample-to-group*
		sgpd					8.9.3	*sample group description*
		subs					8.7.7	*sub-sample information*
		saiz					8.7.8	*sample auxiliary information sizes*
		saio					8.7.9	*sample auxiliary information offsets*
		tfdt					8.8.12	*track fragment decode time*
		meta					8.11.1	*metadata*
		udta					8.10.1	*user-data*
	udta						8.10.1	*user-data*
mfra							8.8.9	*movie fragment random access*
	tfra						8.8.10	*track fragment random access*
	mfro					*	8.8.11	*movie fragment random access offset*
mdat							8.2.2	*media data container*
free							8.1.2	*free space*
skip							8.1.2	*free space*
imda							8.1.4	*media data container that contains an identifier to be used with data references*
meta							8.11.1	*metadata*
	hdlr					*	8.4.3	*handler, declares the metadata (handler) type*
	dinf						8.7.1	*data information box, container*
		dref					8.7.2	*data reference box, declares source(s) of metadata items*
	iloc						8.11.2.3	*item location*
	ipro						8.11.5	*item protection*
		sinf					8.12.2	*protection scheme information box*
			frma				8.12.3	*original format box*
			schm				8.12.6	*scheme type box*
			schi				8.12.7	*scheme information box*
	iinf						8.11.6	*item information*

Table 1 *(continued)*

Box types, structure, and cross-reference								
	`xml`						8.11.2	*XML container*
	`bxml`						8.11.2	*binary XML container*
	`pitm`						8.11.4	*primary item reference*
	`fiin`						8.13.2	*file delivery item information*
		`paen`					8.13.2	*partition entry*
			`fire`				8.13.7	*file reservoir*
			`fpar`				8.13.3	*file partition*
			`fecr`				8.13.4	*FEC reservoir*
		`segr`					8.13.5	*file delivery session group*
		`gitn`					8.13.6	*group id to name*
	`idat`						8.11.11	*item data*
	`iref`						8.11.12	*item reference*
`styp`							8.16.2	*segment type*
`sidx`							8.16.3	*segment index*
`ssix`							8.16.4	*subsegment index*
`prft`							8.16.5	*producer reference time*
`!mov`							8.19.6	*compressed movie box*
`!mof`							8.19.7	*compressed movie fragment box*
`!six`							8.19.8	*compressed segment index box*
`!ssx`							8.19.9	*compressed subsegment index box*

6.4 Time structure overview

Tracks identify, in *decoding order*, of a sequence of *samples.* Each sample has a *decoding timestamp* that is computed by adding to the previous sample's decoding timestamp, the previous sample's duration (as given by the values in the `TimeToSampleBox` or the equivalent field in movie fragments). The decoding timestamp of the first sample is defined as being at time zero. This forms the decoding timeline of a track.

In some coding systems (notably video) samples are coded in a different order from their presentation order. In this case, each sample is assigned a *composition time* (as given by the values in the `CompositionOffsetBox` or the equivalent field in movie fragments), which is computed by adding a *composition offset* to the decoding timestamp. This forms the composition timeline, and re-orders the samples into *composition order.*

The *presentation timeline* for each track is formed by the concatenation of sections of the composition timeline by means of explicit or implicit *edit lists*.

The presentation timelines of all the tracks are aligned at their zero point. This forms the presentation timeline for the presentation as a whole. This aligned zero point is the nominal time at which presentation starts and from which movie and track durations are measured.

Composition offsets can be signed or unsigned. When they are unsigned, the composition timeline and the decoding timeline are related, and the decoding and compositions times can be considered as enabling a hypothetical system, that is capable of instant decoding, to manage buffering. When they are signed, the composition timeline and the decoding timeline are disconnected. They can be reconnected using the `CompositionToDecodeBox`, if needed, though many systems do not use decoding timestamps, merely decoding in decoding order.

6.5 Identifiers

The track identifiers used in an ISO file are unique within that file; no two tracks shall use the same identifier. Under unified identifier handling (see E.18) this uniqueness requirement is extended to other identifiers.

The next track identifier value stored in `next_track_ID` in the `MovieHeaderBox` generally contains a value one greater than the largest identifier value, of the set required to be unique, found in the file. This enables easy generation of a unique identifier under most circumstances. However, if this value is equal to ones (32-bit unsigned maxint), then a search for an unused unique identifier is needed for all additions.

6.6 Brand identification

The definitions of the brands that apply to timed presentations are specified in Annex E.

6.7 Uniform resource locators (URLs)

When a file conformant to this document is identified by URL, a URL fragment (if present) shall follow the syntax(es) documented in Annex C.

When files are identified by a MIME type, the syntax specified in IETF RFC 6381[29] and Annex K shall be used for any parameters.

7 Streaming support

Segmented streaming is supported by segments as defined in 3.1.41 and 8.16.

Transformed streaming into streaming protocols is supported by hint tracks as defined in 12.4, with specific formats for some protocols in Clause 9; there is a general overview in A.12.

8 Box structures

8.1 File structure and general boxes

8.1.1 Media data box

8.1.1.1 Definition

Box Type: `'mdat'`
Container: File
Mandatory: No
Quantity: Zero or more

This box contains the media data. In video tracks, this box would contain video frames. A presentation may contain zero or more `MediaDataBox`es. The actual media data follows the type field; its structure is described by the structure-data (see particularly the `SampleTableBox`, subclause 8.5, and the `ItemLocationBox`, subclause 8.11.2.3).

In large presentations, it may be desirable to have more data in this box than a 32-bit size would permit. In this case, the large variant of the size field, above in subclause 4.2, is used.

There may be any number of these boxes in the file (including zero, if all the media data is in other files). The structure-data refers to media data by its absolute offset within the file (see subclause 8.7.5, the `ChunkOffsetBox`); so `MediaDataBox` headers and free space may easily be skipped, and files without any box structure may also be referenced and used.

8.1.1.2 Syntax

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

8.1.1.3 Semantics

`data` is the contained media data

8.1.2 Free space box

8.1.2.1 Definition

Box Types:	`'free'`,`'skip'`
Container: File or other box	
Mandatory: No	
Quantity: Zero or more	

The contents of a `FreeSpaceBox` are irrelevant and may be ignored, or the box deleted, without affecting the presentation. (Care should be exercised when deleting the box, as this may invalidate offsets used to refer to other data, unless this box is after all the media data).

8.1.2.2 Syntax

```
aligned(8) class FreeSpaceBox extends Box(free_type) {
   unsigned int(8) data[];
}
```

8.1.2.3 Semantics

`free_type` shall be `'free'` or `'skip'`.

8.1.3 Progressive download information box

8.1.3.1 Definition

Box Types:	`'pdin'`
Container: File	
Mandatory: No	
Quantity: Zero or One	

The `ProgressiveDownloadInfoBox` aids the progressive download of an ISO file. The box contains pairs of numbers (to the end of the box) specifying combinations of effective file download bitrate in units of bytes/sec and a suggested initial playback delay in units of milliseconds.

A receiving party can estimate the download rate it is experiencing, and from that obtain an upper estimate for a suitable initial delay by linear interpolation between pairs, or by extrapolation from the first or last entry.

It is recommended that the `ProgressiveDownloadInfoBox` be placed as early as possible in files, for maximum utility.

8.1.3.2 Syntax

```
aligned(8) class ProgressiveDownloadInfoBox
      extends FullBox('pdin', version = 0, 0) {
   for (i=0; ; i++) {   // to end of box
      unsigned int(32)  rate;
      unsigned int(32)  initial_delay;
   }
}
```

8.1.3.3 Semantics

`rate` is a download rate expressed in bytes/second

`initial_delay` is the suggested delay to use when playing the file, such that if download continues at the given rate, all data within the file will arrive in time for its use and playback should not need to stall.

8.1.4 Identified media data box

8.1.4.1 Definition

Box Type: `'imda'`
Container: File
Mandatory: No
Quantity: Zero or more

This box contains the media data. Its semantics are the same as those for `MediaDataBox` but it additionally contains an identifier that is used in setting up data references to the contained media data.

8.1.4.2 Syntax

```
aligned(8) class IdentifiedMediaDataBox extends Box('imda') {
   unsigned int(32) imda_identifier;
   bit(8) data[]; // until the end of the box
}
```

8.1.4.3 Semantics

`imda_identifier` shall differ from the `imda_identifier` values of the other

`IdentifiedMediaDataBox`es of the file.

8.2 Movie structure

8.2.1 Movie box

8.2.1.1 Definition

Box Type: `'moov'`
Container: File
Mandatory: Yes
Quantity: Exactly one

The structure-data for a presentation is stored in the single `MovieBox` which occurs at the top-level of a file. Normally this box is close to the beginning or end of the file, though this is not required.

8.2.1.2 Syntax

```
aligned(8) class MovieBox extends Box('moov'){
}
```

8.2.2 Movie header box

8.2.2.1 Definition

Box Type: `'mvhd'`
Container: `MovieBox`
Mandatory: Yes
Quantity: Exactly one

This box defines overall information which is media-independent, and relevant to the entire presentation considered as a whole.

The duration of a movie is usually the duration of the longest track (as documented by the `duration` field of the `TrackHeaderBox`); if any track has an indefinite duration, the movie should also have an indefinite duration. If there are tracks longer than the movie duration, it is recommended that playback terminate at the movie duration (i.e. that the longer tracks be implicitly trimmed to the movie duration); this can happen when two media have frame durations that are not exactly divisible, for example.

8.2.2.2 Syntax

```
aligned(8) class MovieHeaderBox extends FullBox('mvhd', version, 0) {
   if (version==1) {
      unsigned int(64)   creation_time;
      unsigned int(64)   modification_time;
      unsigned int(32)   timescale;
      unsigned int(64)   duration;
   } else { // version==0
      unsigned int(32)   creation_time;
      unsigned int(32)   modification_time;
      unsigned int(32)   timescale;
      unsigned int(32)   duration;
   }
   template int(32)   rate = 0x00010000;   // typically 1.0
   template int(16)   volume = 0x0100;   // typically, full volume
   const bit(16)   reserved = 0;
   const unsigned int(32)[2]   reserved = 0;
   template int(32)[9]   matrix =
      { 0x00010000,0,0,0,0x00010000,0,0,0,0x40000000 };
      // Unity matrix
   bit(32)[6]   pre_defined = 0;
   unsigned int(32)   next_track_ID;
}
```

8.2.2.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1 in this document)

`creation_time` is an integer that declares the creation time of the presentation (in seconds since midnight, Jan. 1, 1904, in UTC time)

`modification_time` is an integer that declares the most recent time the presentation was modified (in seconds since midnight, Jan. 1, 1904, in UTC time)

`timescale` is an integer that specifies the time-scale for the entire presentation; this is the number of time units that pass in one second. For example, a time coordinate system that measures time in sixtieths of a second has a time scale of 60.

`duration` is an integer that declares length of the presentation (in the indicated timescale). This property is derived from the presentation's tracks: the value of this field corresponds to the duration of the longest track in the presentation. If the duration cannot be determined then duration is set to all 1s.

`rate` is a fixed point 16.16 number that indicates the preferred rate to play the presentation; 1.0 (0x00010000) is normal forward playback

`volume` is a fixed point 8.8 number that indicates the preferred playback volume. 1.0 (0x0100) is full volume.

`matrix` provides a transformation matrix for the video; (u,v,w) are restricted here to (0,0,1), hex values (0,0,0x40000000).

`next_track_ID` is a non-zero integer that indicates a value to use for the `track_ID` of the next track to be added to this presentation. Zero is not a valid `track_ID` value. The value of `next_track_ID` shall be larger than the largest `track_ID` in use. If this value is equal to all 1s (32-bit maxint), and a new media track is to be added, then a search must be made in the file for an unused value of `track_ID`.

8.3 Track structure

8.3.1 Track box

8.3.1.1 Definition

Box Type: `'trak'`
Container: `MovieBox`
Mandatory: Yes
Quantity: One or more

This is a container box for a single track of a presentation. A presentation consists of one or more tracks. Each track carries its own temporal and spatial information. Each track will contain its associated `MediaBox`.

Tracks are used for a number of purposes, including: (a) to contain media data (media tracks) and (b) to contain packetization information for streaming protocols (hint tracks).

There shall be at least one media track within a `MovieBox`, and all the media tracks that contributed to the hint tracks shall remain in the file, even if the media data within them is not referenced by the hint tracks; after deleting all hint tracks, the entire un-hinted presentation shall remain.

8.3.1.2 Syntax

```
aligned(8) class TrackBox extends Box('trak') {
}
```

8.3.2 Track header box

8.3.2.1 Definition

Box Type: `'tkhd'`
Container: `TrackBox`
Mandatory: Yes
Quantity: Exactly one

This box specifies the characteristics of a single track. Exactly one `TrackHeaderBox` is contained in a track.

In the absence of an edit list, the presentation of a track starts at the beginning of the overall presentation. An ‘empty’ edit is used to offset the start time of a track (see 8.6.6).

The tracks marked with the `track_in_movie` flag set to 1 are those that are intended by the file writer for direct presentation. Thus a track that is used as input to another track — either before or after decoding — but that is not presented by itself — should have the `track_in_movie` flag set to 0. Tracks having the `track_in_movie` flag set are candidates for playback, regardless of whether they are media tracks or reception hint tracks. A track may be used as input and also have the `track_in_movie` flag set to 1.

If a track is a member of a group that presents alternatives for presentation, either by using the alternate_group field in the track header, or by using the `TrackGroupBox` with a group type that defines alternatives, then either only the preferred or default choice track should have the `track_in_movie` flag set to 1, or all tracks should have the `track_in_movie` flag set to 1 (if no default is to be indicated).

If an `'altr'` entity group contains entities for playing, `track_in_movie` should be equal to 1 in all the tracks of the entity group.

If in a presentation no tracks have `track_in_movie` set, and therefore it appears that there is nothing to present, then a player may enable a track from each group for presentation; derived specifications can give further guidance and/or restrictions.

Tracks that are marked as not enabled (`track_enabled` set to 0) shall be ignored and treated as if not present. Application environments may offer a way to enable/disable tracks at run-time and dynamically alter the state of this flag.

The processing of the pixel data from the output of the decoder to its rendering on the screen is not a conformance point of ISOBMFF. However, several structures enable signaling of such rendering features. They are based on the following assumptions:

1) Under the `'iso3'` brand or brands that share its requirements, the `TrackHeaderBox` `width` and `height` assume that the rendered pixels are square (i.e. that the pixel aspect ratio is 1:1). For other brands, the use of the associated structure for rendering is undetermined.

2) The `VisualSampleEntry` documents the expected size of the pixel buffer needed to receive the codec output, possibly cropped by in-stream structures.

   EXAMPLE If a video codec works only with multiples of 16 pixels per line or column, and if the width and height of the video fed to the encoder is 1000x500, the encoder will typically, internally use 64x32 blocs of pixels, but it is expected to use codec-specific cropping structures, that are not exposed at the ISOBMFF level to output a 1000x500 video. The `width` and `height` of the `VisualSampleEntry` fields in this case will be 1000x500.

3) The `PixelAspectRatioBox` documents the aspect ratio that should be applied to the pixels output by the decoder but does not imply the adjustment. The associated adjustment should be taken care of by setting scaled values in the `TrackHeaderBox`.

   EXAMPLE If prior to the encoder, the pixels are scaled horizontally by a factor of 1/2, at the output of the decoder, the inverse scaling should be applied to present non distorted videos. This is done by setting the `TrackHeaderBox height` equal to the `VisualSampleEntry height` and the `TrackHeaderBox width` equal to the double of the `VisualSampleEntry width`. Additionally, a `PixelAspectRatioBox` with a `hSpacing` twice bigger than its `vSpacing` field may be used, if in-stream structures do not carry that information.

4) Additionally, a `CleanApertureBox` may be provided to further crop the video.

The processing of the decoded pixel is assumed to be as follows:

1) Any cropping documented by a `CleanApertureBox` is applied on the pixels output by the decoder,

2) Then, if a `CleanApertureBox` is present, the cropped image is then scaled horizontally by the factor `TrackHeaderBox.width`/`CleanAperture.width` and vertically by `TrackHeaderBox.height`/`CleanAperture.height`

3) Otherwise, if a `CleanApertureBox` is not present, the decoded image is then scaled horizontally by the factor `TrackHeaderBox.width`/`SampleEntry.width` and vertically by `TrackHeaderBox.height`/`SampleEntry.height`.

   NOTE 1 This operation is called in previous editions "normalization to track dimensions".

4) The `TrackHeaderBox` matrix is then applied

5) All visual tracks are superposed in increasing order of the `TrackHeaderBox.layer` value

6) The `MovieHeaderBox` matrix is then applied to the composition.

   NOTE 2 This is a theoretical processing model and concrete implementations following it should avoid resampling the image, in particular when the combination of the above operations results in the identity transformation.

The duration field here does not include the duration of following movie fragments, if any, but only of the media in the enclosing `MovieBox`. The `MovieExtendsHeaderBox` may be used to document the duration including movie fragments, when desired and possible.

8.3.2.2 Syntax

```
aligned(8) class TrackHeaderBox
   extends FullBox('tkhd', version, flags){
   if (version==1) {
      unsigned int(64)   creation_time;
      unsigned int(64)   modification_time;
      unsigned int(32)   track_ID;
      const unsigned int(32)   reserved = 0;
      unsigned int(64)   duration;
   } else { // version==0
      unsigned int(32)   creation_time;
      unsigned int(32)   modification_time;
      unsigned int(32)   track_ID;
      const unsigned int(32)   reserved = 0;
      unsigned int(32)   duration;
   }
   const unsigned int(32)[2]   reserved = 0;
   template int(16) layer = 0;
   template int(16) alternate_group = 0;
   template int(16)   volume = {if track_is_audio 0x0100 else 0};
   const unsigned int(16)   reserved = 0;
   template int(32)[9]   matrix=
      { 0x00010000,0,0,0,0x00010000,0,0,0,0x40000000 };
      // unity matrix
   unsigned int(32) width;
   unsigned int(32) height;
}
```

8.3.2.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1 in this document)

`flags` is a 24-bit integer with flags; the following values are defined:

`track_enabled`: Flag mask is 0x000001. The value 1 indicates that the track is enabled. A disabled track (when the value of this flag is zero) is treated as if it were not present.

`track_in_movie`: Flag mask is 0x000002. The value 1 indicates that the track, or one of its alternatives (if any) forms a direct part of the presentation. The value 0 indicates that the track does not represent a direct part of the presentation.

`track_in_preview`: Flag mask is 0x000004. This flag currently has no assigned meaning, and the value should be ignored by readers. In the absence of further guidance (e.g. from derived specifications), the same value as for track_in_movie should be written.

`track_size_is_aspect_ratio`: Flag value is 0x000008. The value 1 indicates that the width and height fields are not expressed in pixel units. The values have the same units but these units are not specified. The values are only an indication of the desired aspect ratio. If the aspect ratios of this track and other related tracks are not identical, then the respective positioning of the tracks is undefined, possibly defined by external contexts.

`creation_time` is an integer that declares the creation time of this track (in seconds since midnight, Jan. 1, 1904, in UTC time).

`modification_time` is an integer that declares the most recent time the track was modified (in seconds since midnight, Jan. 1, 1904, in UTC time).

`track_ID` is an integer that uniquely identifies this track over the entire life-time of this presentation; `track_ID`s are never re-used and cannot be zero.

`duration` is an integer that indicates the duration of this track (in the timescale indicated in the `MovieHeaderBox`) This duration field may be indefinite (all 1s) when either there is no edit list and the `MediaHeaderBox` duration is indefinite (i.e. all 1s), or when an indefinitely repeated edit list is desired (see subclause 8.6.6 for repeated edits).. If there is no edit list and the duration is not indefinite, then the duration shall be equal to the media duration given in the `MediaHeaderBox`, converted into the timescale in the MovieHeaderBox. Otherwise the value of this field is equal to the sum of the durations of all of the track's edits (possibly including repetitions).

layer specifies the front-to-back ordering of video tracks; tracks with lower numbers are closer to the viewer. 0 is the normal value, and -1 would be in front of track 0, and so on.

alternate_group is an integer that specifies a group or collection of tracks. If this field is 0 there is no information on possible relations to other tracks. If this field is not 0, it should be the same for tracks that contain alternate data for one another and different for tracks belonging to different such groups. Only one track within an alternate group should be played or streamed at any one time, and shall be distinguishable from other tracks in the group via attributes such as bitrate, codec, language, packet size etc. A group may have only one member.

volume is a fixed 8.8 value specifying the track's relative audio volume. Full volume is 1.0 (0x0100) and is the normal value. Its value is irrelevant for a purely visual track. Tracks may be composed by combining them according to their volume, and then using the overall MovieHeaderBox volume setting; or more complex audio composition (e.g. MPEG-4 BIFS) may be used.

matrix provides a transformation matrix for the video; (u,v,w) are restricted here to (0,0,1), hex (0,0,0x40000000).

width and height fixed-point 16.16 values are track-dependent as follows:

For text and subtitle tracks, they may, depending on the coding format, describe the suggested size of the rendering area. For such tracks, the value 0x0 may also be used to indicate that the data may be rendered at any size, that no preferred size has been indicated and that the actual size may be determined by the external context or by reusing the width and height of another track. For those tracks, the flag track_size_is_aspect_ratio may also be used.

For non-visual tracks (e.g. audio), they should be set to zero.

For all other tracks, they specify the track's visual presentation size. These need not be the same as the pixel dimensions of the images, which is documented in the sample description(s); all images in the sequence are scaled to this size, before any overall transformation of the track represented by the matrix. The pixel dimensions of the images are the default values.

8.3.3 Track reference box

8.3.3.1 Definition

Box Type: 'tref'
Container: TrackBox
Mandatory: No
Quantity: Zero or one

This box includes a set of TrackReferenceTypeBoxes, each of which indicates, by its type, that the enclosing track has one of more references of that type. Each reference type shall occur at most once. Within each TrackReferenceTypeBox there is an array of track_IDs; within a given array, a given value shall occur at most once. Other structures in the file formats index through these arrays; index values start at 1.

Exactly one TrackReferenceBox can be contained within the TrackBox.

If this box is not present, the track is not referencing any other track in any way. The reference array is sized to fill the reference type box.

8.3.3.2 Syntax

```
aligned(8) class TrackReferenceBox extends Box('tref') {
   TrackReferenceTypeBox [];
}
aligned(8) class TrackReferenceTypeBox (unsigned int(32) reference_type) extends
Box(reference_type) {
   unsigned int(32) track_IDs[];
}
```

8.3.3.3 Semantics

The `TrackReferenceBox` contains `TrackReferenceTypeBox`es. There shall be at most one `TrackReferenceTypeBox` of a given type in a `TrackReferenceBox`.

`track_IDs` is an array of integers providing the track identifiers of the referenced tracks or `track_group_id` values of the referenced track groups. Each value `track_IDs[i]`, where `i` is a valid index to the `track_IDs[]` array, is an integer that provides a reference from the containing track to the track with `track_ID` equal to `track_IDs[i]` or to the track group with both `track_group_id` equal to `track_IDs[i]` and (`flags` & 1) of `TrackGroupTypeBox` equal to 1. When a `track_group_id` value is referenced, the track reference applies to each track of the referenced track group individually unless stated otherwise in the semantics of particular track reference types. The value 0 shall not be present. In the array there shall be no duplicated value; however, a `track_ID` may appear in the array and also be a member of one or more track groups for which the `track_group_ID`s appear in the array. This means that in forming the list of tracks, after replacing `track_group_ID`s by the `track_ID`s of the tracks in those groups, there might be duplicate `track_ID`s. A `track_group_ID` shall not be used when the semantics of the reference requires that the reference be to a single track.

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'` links a descriptive or metadata track to the content which it describes

— `'font'` this track uses fonts carried/defined in the referenced track.

— `'hind'` indicates that the referenced track(s) may contain media data required for decoding of the track containing the track reference, i.e., it should only be used if the referenced hint track is used. The referenced tracks shall be hint tracks. The `'hind'` dependency can, for example, be used for indicating the dependencies between hint tracks documenting layered IP multicast over RTP.

— `'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.

— `'subt'` this track contains subtitle, timed text or overlay graphical information for the referenced track or any track in the alternate group to which the track belongs, if any.

— `'thmb'`: this track contains thumbnail images for the referenced track. A thumbnail track shall not be linked to another thumbnail track with the `'thmb'` item reference.

— `'auxl'`: this track contains auxiliary media for the indicated track (e.g. depth map or alpha plane for video).

— `'cdtg'`: describes the referenced media tracks and track groups collectively; the `'cdtg'` track reference shall only be present in timed metadata tracks.

— `'shsc'`: links a shadow sync track to a main track; see subclause 8.6.3

NOTE 1 A track with reference type 'auxl' could have a coding dependency; its use is clarified by specifications that use it.

NOTE 2 When multiple track references would describe an auxiliary video track, derived specifications might constrain or recommend which track references are used. For example, derived specifications might constrain or recommend whether to use 'vdep' or 'auxl' or both for auxiliary depth video track.

NOTE 3 Other structures index through the array of track references and hence position and order of them can be significant.

NOTE 4 A timed metadata track containing 'cdsc' track reference to a track_group_id value describes each track in the track group individually.

8.3.4 Track group box

8.3.4.1 Definition

Box Type: 'trgr'
Container: TrackBox
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 TrackGroupBox 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 TrackReferenceBox is used for such purposes.

(flags & 1) equal to 1 in a TrackGroupTypeBox of a particular track_group_type indicates that track_group_id in that TrackGroupTypeBox is not equal to any track_ID value and is not equal to track_group_id of any other TrackGroupTypeBox with a different track_group_type. When (flags & 1) is equal to 1 in a TrackGroupTypeBox with particular values of track_group_type and track_group_id, (flags & 1) shall be equal to 1 in all TrackGroupTypeBoxes of the same values of track_group_type and track_group_id, respectively.

8.3.4.2 Syntax

```
aligned(8) class TrackGroupBox extends Box('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. Specified in 8.3.4.4.1.

'ster' indicates that this track is either the left or right view of a stereo pair suitable for playback on a stereoscopic display. Specified in 8.3.4.4.2.

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

8.3.4.4 Track group definitions

8.3.4.4.1 Multi-source presentation

track_group_type equal to 'msrc' indicates that this track belongs to a multi-source presentation. The tracks that have the same value of track_group_id within a TrackGroupTypeBox 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.

8.3.4.4.2 Stereoscopic pair

8.3.4.4.2.1 Definition

TrackGroupTypeBox with track_group_type equal to 'ster' indicates that this track is either the left or right view of a stereo pair suitable for playback on a stereoscopic display.

The tracks that have the same value of track_group_id within StereoVideoGroupBox form a stereo pair, and there shall be no more than two of such tracks for the same value of track_group_id.

NOTE Usually there are two tracks indicated to be a stereo pair with the StereoVideoGroupBox having the same value of track_group_id. However, only one track can be associated with a stereo pair in specific cases. For example, the file can be edited in a manner that one of the tracks forming a stereo pair gets removed. In another example, only one of the tracks of a stereo pair is selected for transmission, e.g. using DASH.

8.3.4.4.2.2 Syntax

```
aligned(8) class StereoVideoGroupBox extends TrackGroupTypeBox('ster')
{
   unsigned int(1) left_view_flag;
   bit(31) reserved;
}
```

8.3.4.4.2.3 Semantics

left_view_flag equal to 0 indicates the right view of a stereo pair, and left_view_flag equal to 1 indicates the left view of a stereo pair. When there are two tracks with the same value of track_group_id, the value of left_view_flag shall differ.

8.3.5 Track type box

8.3.5.1 Definition

Box Type: 'ttyp'
Container: TrackBox
Mandatory: No
Quantity: Zero or one

The payload of TrackTypeBox has the same syntax as the payload of FileTypeBox. The content of an instance of TrackTypeBox shall be such that it would apply as the content of FileTypeBox, if all other tracks of the file were removed and only the track containing this box, and the tracks it references by means of track references, remained in the file.

NOTE TrackTypeBox can be used in specifying media profiles or track-specific brands.

8.3.5.2 Syntax

```
aligned(8) class TrackTypeBox extends GeneralTypeBox ('ttyp')
{}
```

8.4 Track media structure

8.4.1 Media box

8.4.1.1 Definition

Box Type: `'mdia'`
Container: `TrackBox`
Mandatory: Yes
Quantity: Exactly one

The media declaration container contains all the objects that declare information about the media data within a track.

8.4.1.2 Syntax

```
aligned(8) class MediaBox extends Box('mdia') {
}
```

8.4.2 Media header box

8.4.2.1 Definition

Box Type: `'mdhd'`
Container: `MediaBox`
Mandatory: Yes
Quantity: Exactly one

The media header declares overall information that is media-independent, and relevant to characteristics of the media in a track.

8.4.2.2 Syntax

```
aligned(8) class MediaHeaderBox extends FullBox('mdhd', version, 0) {
   if (version==1) {
      unsigned int(64)   creation_time;
      unsigned int(64)   modification_time;
      unsigned int(32)   timescale;
      unsigned int(64)   duration;
   } else { // version==0
      unsigned int(32)   creation_time;
      unsigned int(32)   modification_time;
      unsigned int(32)   timescale;
      unsigned int(32)   duration;
   }
   bit(1)    pad = 0;
   unsigned int(5)[3]    language;   // ISO-639-2/T language code
   unsigned int(16)    pre_defined = 0;
}
```

8.4.2.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1)

`creation_time` is an integer that declares the creation time of the media in this track (in seconds since midnight, Jan. 1, 1904, in UTC time).

`modification_time` is an integer that declares the most recent time the media in this track was modified (in seconds since midnight, Jan. 1, 1904, in UTC time).

`timescale` is an integer that specifies the number of time units that pass in one second for this media. For example, a time coordinate system that measures time in sixtieths of a second has a time scale of 60.

`duration` is an integer that declares the duration of this media (in the scale of the timescale) and should be the largest composition timestamp plus the duration of that sample. If the duration cannot be determined then duration is set to all 1s.

NOTE The duration of an audio track may be smaller than the duration of the audio samples output by the decoder. This depends on the decoding process. The decoding of the last ISOBMFF sample of a track may produce additional audio samples that are not meant to be rendered.

`language` declares the language code for this media, as a packed three-character code defined in ISO 639-2. Each character is packed as the difference between its ASCII value and 0x60. Since the code is confined to being three lower-case letters, these values are strictly positive.

8.4.3 Handler reference box

8.4.3.1 Definition

Box Type: `'hdlr'`
Container: `MediaBox` or `MetaBox`
Mandatory: Yes
Quantity: Exactly one

This box within a `MediaBox` declares media type of the track, and thus the process by which the media-data in the track is presented. For example, a format for which the decoder delivers video would be stored in a video track, identified by being handled by a video handler. The documentation of the storage of a media format identifies the media type which that format uses.

This box when present within a `MetaBox`, declares the structure or format of the `MetaBox` contents.

There is a general handler for metadata streams of any type; the specific format is identified by the sample entry, as for video or audio, for example.

8.4.3.2 Syntax

```
aligned(8) class HandlerBox extends FullBox('hdlr', version = 0, 0) {
   unsigned int(32)   pre_defined = 0;
   unsigned int(32)   handler_type;
   const unsigned int(32)[3]    reserved = 0;
   utf8string    name;
}
```

8.4.3.3 Semantics

`version` is an integer that specifies the version of this box

`handler_type`

— when present in a `MediaBox`, contains a value as defined in Clause 12, or a value from a derived specification, or registration.

— when present in a `MetaBox`, contains an appropriate value to indicate the format of the `MetaBox` contents. The value `'null'` can be used in the primary `MetaBox` to indicate that it is merely being used to hold resources.

`name` gives a human-readable name for the track type (for debugging and inspection purposes).

8.4.4 Media information box

8.4.4.1 Definition

Box Type: `'minf'`
Container: `MediaBox`
Mandatory: Yes
Quantity: Exactly one

This box contains all the objects that declare characteristic information of the media in the track.

8.4.4.2 Syntax

```
aligned(8) class MediaInformationBox extends Box('minf') {
}
```

8.4.5 Media information header boxes

8.4.5.1 Definition

There is a different media information header for each track type (corresponding to the media handler-type); the matching header shall be present, which may be one of those defined in Clause 12, or one defined in a derived specification.

The type of media header is used is determined by the definition of the media type and shall match the media handler.

8.4.5.2 Null media header box

8.4.5.2.1 Definition

Box Types: `'nmhd'`
Container: `MediaInformationBox`
Mandatory: Yes
Quantity: Exactly one specific media header shall be present

Streams for which no specific media header is identified use a `NullMediaHeaderBox`, as defined here.

8.4.5.2.2 Syntax

```
aligned(8) class NullMediaHeaderBox
   extends FullBox('nmhd', version = 0, flags) {
}
```

8.4.5.2.3 Semantics

`version` - is an integer that specifies the version of this box.

`flags` - is a 24-bit integer with flags (currently all zero).

8.4.6 Extended language tag

8.4.6.1 Definition

Box Type: `'elng'`
Container: `MediaBox`
Mandatory: No
Quantity: Zero or one

The ExtendedLanguageBox represents media language information, and shall contain a code in conformance with IETF BCP 47. It is an optional peer of the MediaHeaderBox, and shall occur after the MediaHeaderBox.

The extended language tag can provide better language information than the language field in the MediaHeaderBox, including information such as region, script, variation, and so on, as parts (or subtags).

The ExtendedLanguageBox is optional, and if it is absent the media language should be used. The extended language tag overrides the media language if they are not consistent.

For best compatibility with earlier players, if an extended language tag is specified, the most compatible language code should be specified in the language field of the MediaHeaderBox (for example, "eng" if the extended language tag is "en-UK"). If there is no reasonably compatible tag, the packed form of 'und' can be used.

8.4.6.2 Syntax

```
aligned(8) class ExtendedLanguageBox extends FullBox('elng', 0, 0) {
   utf8string   extended_language;
}
```

8.4.6.3 Semantics

extended_language contains a IETF BCP 47 compliant language tag string, such as "en-US", "fr-FR", or "zh-CN".

8.5 Sample tables

8.5.1 Sample table box

8.5.1.1 Definition

Box Type: 'stbl'
Container: MediaInformationBox
Mandatory: Yes
Quantity: Exactly one

The sample table contains all the time and data indexing of the media samples in a track. Using the tables here, it is possible to locate samples in time, determine their type (e.g. I-frame or not), and determine their size, container, and offset into that container.

If the track that contains the SampleTableBox references no data, then the SampleTableBox does not need to contain any sub-boxes (this is not a very useful media track).

If the track that the SampleTableBox is contained in does reference data, then the following sub-boxes are required: SampleDescriptionBox, SampleSizeBox (or CompactSampleSizeBox), SampleToChunkBox, and ChunkOffsetBox (or ChunkLargeOffsetBox). Further, the SampleDescriptionBox shall contain at least one entry. A SampleDescriptionBox is required because it contains the data reference index field which indicates which DataEntry to use to retrieve the media samples. Without the SampleDescriptionBox, it is not possible to determine where the media samples are stored. The SyncSampleBox is optional.

The SyncSampleBox should be present in the SampleTableBox if some samples in the track, including any track fragments, are non sync samples, but the flag sample_is_non_sync_sample of samples in track fragments is valid and describes the samples, even if the SyncSampleBox is not present. If the track is not fragmented and the SyncSampleBox is not present, all samples in the track are sync samples.

A.9 provides a narrative description of random access using the structures defined in the SampleTableBox.

8.5.1.2 Syntax

```
aligned(8) class SampleTableBox extends Box('stbl') {
}
```

8.5.2 Sample description box

8.5.2.1 Definition

Box Types: `'stsd'`
Container: `SampleTableBox`
Mandatory: Yes
Quantity: Exactly one

The sample description table gives detailed information about the coding type used, and any initialization information needed for that coding. The syntax of the sample entry used is determined by both the format field and the media handler type.

The information stored in the `SampleDescriptionBox` after the entry-count is both track-type specific as documented here, and can also have variants within a track type (e.g. different codings may use different specific information after some common fields, even within a video track).

Which type of sample entry form is used is determined by the media handler, using a suitable form, such as one defined in Clause 12, or defined in a derived specification, or registration.

Multiple descriptions may be used within a track.

NOTE 1 Though the count is 32 bits, the number of items is usually much fewer, and is restricted by the fact that the reference index in the sample table is only 16 bits

If the ‘format’ field of a `SampleEntry` is unrecognized, neither the sample description itself, nor the associated media samples, shall be decoded.

NOTE 2 The `format` field of a `SampleEntry` is restricted in this document not to contain the character '.' (dot). Derived specifications are encouraged to avoid using it as well as any character that requires encoding of the "codecs" parameter; see Annex K.

Derived specifications deriving Sample Entry classes listed in the table of 8.12.1 should be extremely careful. Derivation by adding boxes at the end of the class should be preferred as it preserves Sample Entry parsing and does not require a new `'encX'` value. Adding a new field to a class will not allow for the use of the associated `'encX'` scheme for parsing reasons. A new `'encX'` scheme will have to be defined for signaling encrypted stream based on that derived class.

The definition of sample entries specifies boxes in a particular order, and this is usually also followed in derived specifications. For maximum compatibility, writers should construct files respecting the order both within specifications and as implied by the inheritance, whereas readers should be prepared to accept any box order.

All `SampleEntry` boxes may contain “extra boxes” not explicitly defined in the box syntax of this or derived specifications. When present, such boxes shall follow all defined fields and should follow any defined contained boxes. Decoders shall presume a sample entry box could contain extra boxes and shall continue parsing as though they are present until the containing box length is exhausted.

An optional `BitRateBox` may be present in any `SampleEntry` to signal the bit rate information of a stream. This can be used for buffer configuration.

All string fields shall be of type `utf8string` and null-terminated, even if unused. “Optional” means there is at least one null byte.

Entries that identify the format by MIME type, such as a `TextSubtitleSampleEntry`, `TextMetaDataSampleEntry`, or `SimpleTextSampleEntry`, all of which contain a MIME type, may be used to identify the format of streams for which a MIME type applies. A MIME type applies if the contents of the string in the optional configuration box (without its null termination), followed by the contents of a set of samples, starting with a sync sample and ending at the sample immediately preceding a sync sample, are concatenated in their entirety, and the result meets the decoding requirements for documents of that MIME type. Non-sync samples should be used only if that format specifies the behaviour of 'progressive decoding', and then the sample times indicate when the results of such progressive decoding should be presented (according to the media type).

NOTE 3 The samples in a track that is all sync samples are therefore each a valid document for that MIME type.

In some classes derived from `SampleEntry`, `namespace` and `schema_location` are used both to identify the XML document content and to declare "brand" or profile compatibility. Multiple namespace identifiers indicate that the track conforms to the specification represented by each of the identifiers, some of which may identify supersets of the features present. A decoder should be able to decode all the namespaces in order to be able to decode and present correctly the media associated with this sample entry.

NOTE 4 Additionally, namespace identifiers might represent performance constraints, such as limits on document size, font size, drawing rate, etc., as well as syntax constraints, such as features that are not permitted or ignored.

8.5.2.2 Syntax

```
aligned(8) abstract class SampleEntry (unsigned int(32) format)
   extends Box(format){
   const unsigned int(8)[6] reserved = 0;
   unsigned int(16) data_reference_index;
}
class BitRateBox extends Box('btrt'){
   unsigned int(32) bufferSizeDB;
   unsigned int(32) maxBitrate;
   unsigned int(32) avgBitrate;
}
aligned(8) class SampleDescriptionBox ()
   extends FullBox('stsd', version, 0){
   int i ;
   unsigned int(32) entry_count;
   for (i = 1 ; i <= entry_count ; i++){
      SampleEntry();      // an instance of a class derived from SampleEntry
   }
}
```

8.5.2.3 Semantics

`version` is set to zero. A version number of 1 shall be treated as a version of 0.

`entry_count` is an integer that gives the number of entries in the following table

`SampleEntry` is the appropriate sample entry.

`data_reference_index` is an integer that contains the index of the `DataEntry` to use to retrieve data associated with samples that use this sample description. Data entries are stored in `DataReferenceBox`es. The index ranges from 1 to the number of data entries.

`bufferSizeDB` gives the size of the decoding buffer for the elementary stream in bytes.

`maxBitrate` gives the maximum rate in bits/second over any window of one second; this is a measured value for stored content, or a value that a stream is configured not to exceed; the stream shall not exceed this bitrate.

`avgBitrate` gives the average rate in bits/second of the stream; this is a measured value (cumulative over the entire presentation) for stored content, or the configured target average bitrate for a stream.

8.5.3 Degradation priority box

8.5.3.1 Definition

Box Type: `'stdp'`
Container: `SampleTableBox`
Mandatory: No.
Quantity: Zero or one.

This box contains the degradation priority of each sample. The values are stored in the table, one for each sample. The size of the table, `sample_count` is taken from the `sample_count` in the `SampleSizeBox`. Specifications derived from this define the exact meaning and acceptable range of the priority field.

8.5.3.2 Syntax

```
aligned(8) class DegradationPriorityBox
   extends FullBox('stdp', version = 0, 0) {
   int i;
   for (i=0; i < sample_count; i++) {
      unsigned int(16)   priority;
   }
}
```

8.5.3.3 Semantics

`version` - is an integer that specifies the version of this box.

`priority` - is integer specifying the degradation priority for each sample.

8.5.4 Sample scale box

This box has been deprecated and is no longer defined in this document.

8.6 Track time structures

8.6.1 Time to sample boxes

8.6.1.1 Definition

The composition timestamps (CT) and decoding timestamps (DT) of samples are derived from the time to sample boxes, of which there are two types. The decoding timestamp is defined by the `TimeToSampleBox`, which documents the sample duration, that is, the difference between the decoding timestamp of the following sample and the sample at hand. The composition timestamps are defined in the `CompositionOffsetBox` as time offsets from decoding timestamps. If the composition and decoding timestamps are identical for every sample in the track, then only the `TimeToSampleBox` is required; the `CompositionOffsetBox` shall not be present and all composition offsets are defined to be zero.

The `TimeToSampleBox` shall give non-zero durations for all samples with the possible exception of the last one. Durations in the `TimeToSampleBox` are strictly positive (non-zero), except for the very last entry, which may be zero. This rule derives from the rule that there shall not be two samples in a stream with the same decoding timestamp. Great care must be taken when adding samples to a stream, that the sample that was previously last may need to have a non-zero duration established, in order to observe this rule. One approach in the case where the the duration of the last sample is indeterminate is to use an arbitrary small value and a 'dwell' edit.

Some coding systems may allow samples that are used only for reference and not output (e.g. a non-displayed reference frame in video). When any such non-output sample is present in a track, the following applies:

1) A non-output sample shall be given a composition timestamp which is outside the time-range of the samples that are output.

2) An edit list shall be used to exclude the composition times of the non-output samples.

3) When the track includes a `CompositionOffsetBox`,

   a) version 1 of the `CompositionOffsetBox` shall be used,

   b) the value of `sample_offset` shall be set equal to the most negative number possible (for 32-bit values, -2^{31}) for each non-output sample,

   c) the `CompositionToDecodeBox` should be contained in the `SampleTableBox` of the track, and

   d) when the `CompositionToDecodeBox` is present for the track, the value of `leastDecodeToDisplayDelta` field in the box shall be equal to the smallest composition offset in the `CompositionOffsetBox` excluding the `sample_offset` values for non-output samples.

NOTE Thus, `leastDecodeToDisplayDelta` is greater than -2^{31}.

In the example in Table 2 and Table 3, there is a sequence of I, P, and B frames, each with a sample duration of 10. The samples are stored as follows, with the indicated values for their sample durations and composition time offsets (the actual composition timestamp, CT, and decoding timestamp, DT, are given for reference). The re-ordering occurs because the predicted P frames must be decoded before the bi-directionally predicted B frames. The value of DT for a sample is always the sum of the durations of the preceding samples. Thus, in the absence of composition offsets, the total of the sample durations is the duration of the media in this track.

Table 2 — Closed GOP example

GOP	/--	---	---	---	---	---	--\	/--	---	---	---	---	---	--\
	I1	P4	B2	B3	P7	B5	B6	I8	P11	B9	B10	P14	B12	B13
DT	0	10	20	30	40	50	60	70	80	90	100	110	120	130
CT	10	40	20	30	70	50	60	80	110	90	100	140	120	130
Duration	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Composition offset	10	30	0	0	30	0	0	10	30	0	0	30	0	0

Table 3 — Open GOP example

GOP	/--	--	--	--	--	--\	/-	--	--	--	---	--\
	I3	B1	B2	P6	B4	B5	I9	B7	B8	P12	B10	B11
DT	0	10	20	30	40	50	60	70	80	90	100	110
CT	30	10	20	60	40	50	90	70	80	120	100	110
Duration	10	10	10	10	10	10	10	10	10	10	10	10
Composition offset	30	0	0	30	0	0	30	0	0	30	0	0

8.6.1.2 Decoding time to sample box

8.6.1.2.1 Definition

Box Type: `'stts'`
Container: `SampleTableBox`

Mandatory: Yes
Quantity: Exactly one

This box contains a compact version of a table that allows indexing from decoding timestamp to sample number. Other tables give sample sizes and pointers, from the sample number. Each entry in the table gives the number of consecutive samples with the same sample duration, and that sample duration. By adding the sample durations a complete time-to-sample map may be built.

The `TimeToSampleBox` contains sample durations, the differences in decoding timestamps (DT):

DT[n+1] = DT[n] + `sample_delta`[n]

The sample entries are ordered by decoding timestamps; therefore all the values of `sample_delta` shall be non-negative.

The DT axis has a zero origin; DT[i] = SUM[for j=0 to i-1 of `sample_delta`[j]], and in the absence of composition offsets, the sum of all sample durations gives the duration of the media in the track (not mapped to the overall timescale, and not considering any edit list).

8.6.1.2.2 Syntax

```
aligned(8) class TimeToSampleBox
   extends FullBox('stts', version = 0, 0) {
   unsigned int(32)   entry_count;
      int i;
   for (i=0; i < entry_count; i++) {
      unsigned int(32)   sample_count;
      unsigned int(32)   sample_delta;
   }
}
```

For example with Table 2, the entry would be:

Sample count	Sample-delta
14	10

8.6.1.2.3 Semantics

`version` - is an integer that specifies the version of this box.

`entry_count` - is an integer that gives the number of entries in the following table.

`sample_count` - is an integer that counts the number of consecutive samples that have the given duration.

`sample_delta` - is an integer that gives the difference between the decoding timestamp of the next sample and this one, in the time-scale of the media.

8.6.1.3 Composition time to sample box

8.6.1.3.1 Definition

Box Type: `'ctts'`
Container: `SampleTableBox`
Mandatory: No
Quantity: Zero or one

This box provides the offset between decoding timestamp and composition timestamp. In version 0 of this box the decoding timestamp must be less than the composition timestamp, and the offsets are expressed as unsigned numbers such that if CT is the composition timestamp, CT[n] = DT[n] + `sample_offset`[n].

In version 1 of this box, the composition timestamp is still derived from the decoding timestamp, but the offsets are signed. It is recommended that for the computed composition timestamps, there is exactly one sample with the value 0 (zero), and that no sample have a composition timestamp less than zero.

NOTE the presentation of samples with a composition timestamp less than 0 is undefined, as time 0 is the start time of the presentation across all tracks; nor can edit lists refer to samples with such composition timestamps.

Composition cannot happen before actual decoding. If negative offsets are used such that the composition timestamp of a sample becomes smaller than its decoding timestamp: either the decoding timestamp is ignored (for systems that only need decoding order, for example) or if decoding timestamps are needed, the decoding timeline must be offset to ensure that decoding happens in time. The `CompositionToDecodeBox` can be used to give advice on what offset may be needed.

For either version of the box, each sample shall have a unique composition timestamp, that is, the composition timestamp for two samples in the same track 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 timestamp of 0, an edit list may be used to 'shift' the media to time 0.

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

For example in Table 2

Sample count	Sample_offset
1	10
1	30
2	0
1	30
2	0
1	10
1	30
2	0
1	30
2	0

8.6.1.3.2 Syntax

```
aligned(8) class CompositionOffsetBox
   extends FullBox('ctts', version, 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;
```

```
        }
    }
}
```

8.6.1.3.3 Semantics

`version` - is an integer that specifies the version of this box.

`entry_count` is an integer that gives the number of entries in the following table.

`sample_count` is an integer that counts the number of consecutive samples that have the given offset.

`sample_offset` is an integer that gives the offset between CT and DT, such that CT[n] = DT[n]+ `sample_offset`[n].

8.6.1.4 Composition to decode box

8.6.1.4.1 Definition

Box Type: `'cslg'`
Container: `SampleTableBox` or `TrackExtensionPropertiesBox`
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. This box is only advisory; it documents values that could be calculated by inspecting the track. There is no normative processing associated with this box. Readers that want to assure that, for example, all decoding timestamps precede composition timestamps might subtract a value equal to or larger than `compositionToDTSShift` from the decoding timestamps in the stream (note that this may yield negative decoding timestamps).

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 timestamp 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.

When the `CompositionToDecodeBox` is included in the `SampleTableBox`, it documents the composition and decoding time relationships of the samples in the `MovieBox` only, not including any subsequent movie fragments. When the `CompositionToDecodeBox` is included in the `TrackExtensionPropertiesBox`, it documents the composition and decoding time relationships of the samples in all movie fragments following the `MovieBox`.

Version 1 of this box supports 64-bit times and should only be used if needed (at least one value does not fit into 32 bits).

NOTE in the absence of this box when signed composition offsets are used, correct decoding timestamps cannot in general be re-computed, even with complete inspection of all the samples. Hence, in order to enable converting ISOBMFF content to formats that do not support negative composition offsets, the `CompositionToDecodeBox` may be necessary.

8.6.1.4.2 Syntax

```
class CompositionToDecodeBox extends FullBox('cslg', version, 0) {
   if (version==0) {
      signed int(32)    compositionToDTSShift;
      signed int(32)   leastDecodeToDisplayDelta;
      signed int(32)   greatestDecodeToDisplayDelta;
      signed int(32)   compositionStartTime;
      signed int(32)   compositionEndTime;
   } else {
      signed int(64)   compositionToDTSShift;
      signed int(64)   leastDecodeToDisplayDelta;
      signed int(64)   greatestDecodeToDisplayDelta;
      signed int(64)   compositionStartTime;
      signed int(64)   compositionEndTime;
   }
}
```

8.6.1.4.3 Semantics

`compositionToDTSShift`: if this value is added to the composition timestamps (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 `CompositionOffsetBox` in this track

`greatestDecodeToDisplayDelta`: the largest composition offset in the `CompositionOffsetBox` in this track

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

`compositionEndTime`: the composition timestamp plus the composition duration, of the sample with the largest computed composition timestamp (CTS) in the media of this track; if this field takes the value 0, the composition end time is unknown.

8.6.2 Sync sample box

8.6.2.1 Definition

Box Type: `'stss'`
Container: `SampleTableBox`
Mandatory: No
Quantity: Zero or one

This box provides a compact marking of sync samples within the stream. The table is arranged in strictly increasing order of sample number.

If the `SyncSampleBox` is not present, every sample is a sync sample.

NOTE it is not required that every sync sample be marked by this table (or the equivalent flag in Movie Fragments), only that samples so marked actually be sync samples.

8.6.2.2 Syntax

```
aligned(8) class SyncSampleBox
   extends FullBox('stss', version = 0, 0) {
   unsigned int(32)   entry_count;
   int i;
   for (i=0; i < entry_count; i++) {
      unsigned int(32)   sample_number;
   }
}
```

8.6.2.3 Semantics

`version` - is an integer that specifies the version of this box.

`entry_count` is an integer that gives the number of entries in the following table. If entry_count is zero, there are no sync samples within the stream and the following table is empty.

`sample_number` gives, for each sync sample in the stream, its sample number.

8.6.3 Shadow sync

8.6.3.1 Shadow sync support

There are two forms of support for shadow sync; the sample table box in this subclause, and the use of the shadow sync track reference type in 8.3.3.

A track containing an `'shsc'` track reference is called a shadow sync sample track, and the tracks pointed to by the `'shsc'` track reference are called main tracks.

The shadow sync sample track provides an optional set of sync samples that can be used when seeking to a position or for similar operations performed to any of the associated main tracks.

When an `'shsc'` track reference is present, the following constraints shall be obeyed:

— All samples of the shadow sync sample track shall be sync samples.

— Each main track shall have a sample that is aligned in decoding time with each sample of the shadow sync sample track.

— A concatenation of the following samples in the following order shall conform to the sample entry of the main track:

  — Any selected sample of the shadow sync sample track, with the sample duration of the sample of the main track that is aligned in decoding time with the selected sample of the shadow sync sample track.

  — Samples of the main track following the sample of the main track that is aligned in decoding time with the selected sample of the shadow sync sample track.

An `'shsc'` track reference indicates that the decoded samples resulting from the concatenation specified above have acceptable quality for playback.

NOTE The samples in the main track that are aligned in decoding time with the samples in the shadow sync sample track are "switchable" samples that are constrained so that no samples preceding a "switchable" sample in decoding order are used as a prediction reference for any sample following the "switchable" sample in decoding order.

8.6.3.2 Shadow sync sample box

8.6.3.2.1 Definition

Box Type: `'stsh'`
Container: `SampleTableBox`
Mandatory: No
Quantity: Zero or one

The shadow sync table provides an optional set of sync samples that can be used when seeking or for similar purposes. In normal forward play they are ignored.

Each entry in the `ShadowSyncSampleBox` consists of a pair of sample numbers. The first entry (shadowed-sample-number) indicates the number of the sample that a shadow sync will be defined for. This should always be a non-sync sample (e.g. a frame difference). The second sample number (sync-sample-number) indicates the sample number of the sync sample (i.e. key frame) that can be used when there is a need for a sync sample at, or before, the shadowed-sample-number.

The entries in the `ShadowSyncSampleBox` shall be sorted based on the shadowed-sample-number field.

The shadow sync samples are normally placed in an area of the track that is not presented during normal play (edited out by means of an edit list), though this is not a requirement. The shadow sync table can be ignored and the track will play (and seek) correctly if it is ignored (though perhaps not optimally).

The ShadowSyncSample replaces, not augments, the sample that it shadows (i.e. the next sample sent is shadowed-sample-number+1). The shadow sync sample is treated as if it occurred at the time of the sample it shadows, having the duration of the sample it shadows.

Hinting and transmission might become more complex if a shadow sample is used also as part of normal playback, or is used more than once as a shadow. In this case the hint track might need separate shadow syncs, all of which can get their media data from the one shadow sync in the media track, to allow for the different timestamps etc. needed in their headers.

8.6.3.2.2 Syntax

```
aligned(8) class ShadowSyncSampleBox
   extends FullBox('stsh', version = 0, 0) {
   unsigned int(32)   entry_count;
   int i;
   for (i=0; i < entry_count; i++) {
      unsigned int(32)   shadowed_sample_number;
      unsigned int(32)   sync_sample_number;
   }
}
```

8.6.3.2.3 Semantics

`version` - is an integer that specifies the version of this box.

`entry_count` - is an integer that gives the number of entries in the following table.

`shadowed_sample_number` - gives the number of a sample for which there is an alternative sync sample.

`sync_sample_number` - gives the number of the alternative sync sample.

8.6.4 Independent and disposable samples box

8.6.4.1 Definition

Box Types: `'sdtp'`
Container: `SampleTableBox`
Mandatory: No
Quantity: Zero or one

This optional table answers three questions about sample dependency:

1) does this sample depend on others (e.g. is it an I-picture)?

2) do no other samples depend on this one?

3) does this sample contain multiple (redundant) encodings of the data at this time-instant (possibly with different dependencies)?

In the absence of this table:

1) the sync sample table (partly) answers the first question; in most video codecs, I-pictures are also sync points,

2) the dependency of other samples on this one is unknown.

3) the existence of redundant coding is unknown.

When performing ‘trick’ modes, such as fast-forward, it is possible to use the first piece of information to locate independently decodable samples. Similarly, when performing random access, it may be necessary to locate the previous sync sample or random access recovery point, and roll-forward from the sync sample or the pre-roll starting point of the random access recovery point to the desired point. While rolling forward, samples on which no others depend need not be retrieved or decoded.

The value of `sample_is_depended_on` is independent of the existence of redundant codings. However, a redundant coding may have different dependencies from the primary coding; if redundant codings are available, the value of `sample_depends_on` documents only the primary coding.

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 timestamp 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 shall therefore not be marked as having no dependency.

For tracks with a `handler_type` that is not `'vide'`, `'soun'`, `'hint'` or `'auxv'`, if another sample with `sample_depends_on=2` or another sample tagged as a “Sync Sample” has already been processed and unless specified otherwise, a sample tagged with `sample_depends_on=2` and `sample_has_redundancy=1` can be discarded, and its duration added to the duration of the preceding one, to maintain the timing of subsequent samples.

The size of the table, `sample_count`, is taken from the `sample_count` in the `SampleSizeBox` or `CompactSampleSizeBox`.

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;
   }
}
```

8.6.4.3 Semantics

`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);

`sample_depends_on` takes one of the following four values:

0: the dependency of this sample is unknown;

1: this sample does depend on others (not an I picture);

2: this sample does not depend on others (I picture);

3: reserved

`sample_is_depended_on` takes one of the following four values:

0: the dependency of other samples on this sample is unknown;

1: other samples may depend on this one (not disposable);

2: no other sample depends on this one (disposable);

3: reserved

`sample_has_redundancy` takes one of the following four values:

0: it is unknown whether there is redundant coding in this sample;

1: there is redundant coding in this sample;

2: there is no redundant coding in this sample;

3: reserved

8.6.5 Edit box

8.6.5.1 Definition

Box Type: `'edts'`
Container: `TrackBox`
Mandatory: No
Quantity: Zero or one

An `EditBox` maps the presentation timeline to the media timeline as it is stored in the file. The `EditBox` is a container for the edit lists.

The `EditBox` is optional. In the absence of this box, there is an implicit one-to-one mapping of these timelines, and the presentation of a track starts at the beginning of the presentation. An empty edit is used to offset the start time of a track.

8.6.5.2 Syntax

```
aligned(8) class EditBox extends Box('edts') {
}
```

8.6.6 Edit list box

8.6.6.1 Definition

Box Type: `'elst'`
Container: `EditBox`
Mandatory: No
Quantity: Zero or one

This box contains an explicit timeline map. Each entry defines part of the track timeline: by mapping part of the composition timeline, or by indicating ‘empty’ time (portions of the presentation timeline that map to no media, an 'empty' edit), or by defining a 'dwell', where a single time-point in the media is held for a period.

NOTE 1 Edits are not restricted to fall on sample times. This means that when entering an edit, it can be necessary to (a) back up to a sync point, and pre-roll from there and then (b) be careful about the duration of the first sample — it might have been truncated if the edit enters it during its normal duration. If this is audio, that frame might need to be decoded, and then the final slicing done. Likewise, the duration of the last sample in an edit might need slicing. The length of the whole track in an `EditListBox` might be the overall duration of the whole movie excluding fragments of a fragment movie. Since edit lists cannot occur in movie fragments, there is an implied edit at the end of the current explicit or implied edit list, that inserts the new media material and the presentation of fragments starts after the presentation of the movie in the `MovieBox`.

Starting offsets for tracks (streams) are represented by an initial 'empty' edit.

A non-'empty' edit may insert a portion of the media timeline that is not present in the initial movie, and is present only in subsequent movie fragments.

Edit lists may be repeated; this is indicated with the `RepeatEdits` flag. When this flag is equal to 0 the edit list is not repeated, while the value 1 specifies that the edit list is repeated. When an `EditListBox` indicates the playback of zero samples or one sample, `RepeatEdits` shall be equal to 0. When the `TrackHeaderBox` duration is not indefinite (all 1s), then the edit list is repeated R times such that the total duration of the edit list multiplied by R equals the `TrackHeaderBox` duration (R is not necessarily an integer). If the `TrackHeaderBox` duration is indefinite, then the edit list is repeated indefinitely.

NOTE 2 When the edit list is repeated, media at time 0 resulting from the edit list follows immediately the media having the largest time resulting from the edit list. In other words, the edit list is repeated seamlessly.

When a movie is fragmented, and does not contain any `MovieExtendsHeaderBox`, the last entry in a track edit list may be a non-empty entry with a media duration field set to 0; in this case, readers shall interpret this media duration as being the accumulated duration of all samples defined in the initial movie and any further movie-fragments. If there is a `MovieExtendsHeaderBox`, the last entry in the edit list should be adjusted such that the total duration of all edits corresponds to the duration in the `MovieExtendsHeaderBox`.

It is recommended that such an edit be used to establish a presentation time of 0 for the first presented sample, when composition offsets are used.

In the case of a fragmented movie, starting offsets for tracks (streams) are also represented by an initial 'empty' edit, followed by a non-'empty' edit.

NOTE 3 The field `edit_duration` used to be called `segment_duration` in previous editions of this document.

8.6.6.2 Syntax

```
aligned(8) class EditListBox extends FullBox('elst', version, flags) {
   unsigned int(32)   entry_count;
   for (i=1; i <= entry_count; i++) {
      if (version==1) {
         unsigned int(64) edit_duration;
         int(64) media_time;
      } else { // version==0
         unsigned int(32) edit_duration;
         int(32)    media_time;
      }
      int(16) media_rate_integer;
      int(16) media_rate_fraction;
   }
}
```

8.6.6.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1)

`flags` the following values are defined. The values of `flags` greater than 1 are reserved.

RepeatEdits   1

entry_count is an integer that gives the number of entries in the following table

edit_duration is an integer that specifies the duration of this edit in units of the timescale in the MovieHeaderBox

media_time is an integer containing the starting time within the media of this edit entry (in media time scale units, in composition time). If this field is set to –1, it is an empty edit. The last edit in a track shall never be an empty edit. Any difference between the duration in the MovieHeaderBox, and the track's duration is expressed as an implicit empty edit at the end.

media_rate specifies the relative rate at which to play the media corresponding to this edit entry. If this value is 0, then the edit is specifying a 'dwell': the media at media-time is presented for the edit_duration. This is expressed as a 16.16 fixed-point integer (16 bits each for the integer and fractional part). The normal value, indicating normal-speed forward play, is 1.0 (integer part equal to 1, fraction part equal to 0).

8.6.6.4 Edit list examples

To play a track from its start for 30 seconds, but at 10 seconds into the presentation, we have the following edit list:

entry_count = 2

edit_duration = 10 seconds
media_time = -1
media_rate = 1

edit_duration = 30 seconds (could be the length of the whole track)
media_time = 0 seconds
media_rate = 1

As an example of correcting for a non-zero initial composition timestamp, if the composition timestamp of the first composed frame is 20, then the edit that maps the media time from 20 onwards to movie time 0 onwards, would read:

entry_count = 1

edit_duration = 0
media_time = 20
media_rate = 1

As an example of an initial offset, to play a track from its start for 0 seconds, but at 2 seconds into the presentation, we have the following edit list:

entry_count = 2
edit_duration = 2 seconds
media_time = -1
media_rate = 1
edit_duration = 0 seconds
media_time = 0 seconds
media_rate = 1

8.7 Track data layout structures

8.7.1 Data information box

8.7.1.1 Definition

Box Type: `'dinf'`
Container: `MediaInformationBox` or `MetaBox`
Mandatory: Yes (required within `MediaInformationBox`) and No (optional within `MetaBox`)
Quantity: Exactly one

The `DataInformationBox` contains objects that declare the location of the media information in a track.

8.7.1.2 Syntax

```
aligned(8) class DataInformationBox extends Box('dinf') {
}
```

8.7.2 Data reference box

8.7.2.1 Definition

Box Type: `'dref'`
Container: `DataInformationBox`
Mandatory: Yes
Quantity: Exactly one

Box Types: `'url '`, `'urn '`
Container: `DataReferenceBox`
Mandatory: Yes (at least one of `'url '` or `'urn '` shall be present)
Quantity: One or more

Box Type: `'imdt'`
Container: `DataReferenceBox`
Mandatory: No.
Quantity: Zero or more.

Box Type: `'snim'`
Container: `DataReferenceBox`
Mandatory: No.
Quantity: Zero or more.

The data reference object contains a table of data references (normally URLs) that declare the location(s) of the media data used within the presentation. The data reference index in the sample description ties entries in this table to the samples in the track. A track may be split over several sources in this way.

If the flag is set indicating that the data is in the same file as this box, then no string (not even an empty one) shall be supplied in the entry field.

The entry_count in the `DataReferenceBox` shall be 1 or greater.

NOTE Though the count is 32 bits, the number of items is usually much fewer, and is restricted by the fact that the reference index in the sample table is only 16 bits

When a file that has data entries with the flag set indicating that the media data is in the same file, is split into segments for transport, the value of this flag does not change, as the file is (logically) reassembled after the transport operation.

The `DataEntryImdaBox` identifies the `IdentifiedMediaDataBox` containing the media data accessed through the `data_reference_index` corresponding to this `DataEntryImdaBox`. The `DataEntryImdaBox` contains the value of `imda_identifier` of the referred `IdentifiedMediaDataBox`. The media data offsets are relative to the first byte of the payload of the referred `IdentifiedMediaDataBox`. In other words, media data offset 0 points to the first byte of the payload of the referred `IdentifiedMediaDataBox`.

The `DataEntrySeqNumImdaBox` identifies the `IdentifiedMediaDataBox` containing the media data accessed through the `data_reference_index` corresponding to this `DataEntrySeqNumImdaBox`. When a `data_reference_index` included in a sample entry refers to `DataEntrySeqNumImdaBox`, each sample referring to the sample entry shall be contained in a movie fragment, and media data offset 0 points to the first byte of the payload of the `IdentifiedMediaDataBox` that has `imda_identifier` equal to `sequence_number` of the `MovieFragmentHeaderBox` of the movie fragment containing the sample.

8.7.2.2 Syntax

```
aligned(8) class DataEntryBaseBox(entry_type, bit(24) flags)
   extends FullBox(entry_type, version = 0, flags) {
}
aligned(8) class DataEntryUrlBox (bit(24) flags)
   extends DataEntryBaseBox('url ', flags) {
   utf8string location;
}
aligned(8) class DataEntryUrnBox (bit(24) flags)
   extends DataEntryBaseBox('urn ', flags) {
   utf8string name;
   utf8string location;
}
aligned(8) class DataEntryImdaBox (bit(24) flags)
   extends DataEntryBaseBox('imdt', flags) {
   unsigned int(32) imda_ref_identifier;
}
aligned(8) class DataEntrySeqNumImdaBox (bit(24) flags)
   extends DataEntryBaseBox ('snim', flags) {
}
aligned(8) class DataReferenceBox
   extends FullBox('dref', version = 0, 0) {
   unsigned int(32)   entry_count;
   for (i=1; i <= entry_count; i++) {
      DataEntryBaseBox(entry_type, entry_flags)   data_entry;
   }
}
```

8.7.2.3 Semantics

`version` is an integer that specifies the version of this box

`entry_count` is an integer that counts the actual entries

`entry_flags` is a 24-bit integer with flags; one flag is defined (x000001) which means that the media data is in the same file as the `Box` containing this data reference. If this flag is set, the `DataEntryUrlBox` shall be used and no string is present; the box terminates with the entry-flags field.

`data_entry` is an instance of a class derived from `DataEntryBaseBox`.

`name` is a URN, and is required in a URN entry

`location` is a URL, and is required in a URL entry and optional in a URN entry, where it gives a location to find the resource with the given name. The URL type should be of a service that delivers a file (e.g. URLs of type file, http, ftp etc.), and which services ideally also permit random access. Relative URLs are permissible and are relative to the file that contains this data reference.

`imda_ref_identifier` identifies the `IdentifiedMediaDataBox` containing the media data accessed through the `data_reference_index` corresponding to this `DataEntryImdaBox`. The referred `IdentifiedMediaDataBox` contains `imda_identifier` that is equal to `imda_ref_identifier`.

8.7.3 Sample size boxes

8.7.3.1 Definition

Box Type: `'stsz'`, `'stz2'`
Container: `SampleTableBox`
Mandatory: Yes
Quantity: Exactly one variant shall be present

This box contains the sample count and a table giving the size in bytes of each sample. This allows the media data itself to be unframed. The total number of samples in the media is always indicated in the sample count.

There are two variants of the sample size box. The first variant has a fixed size 32-bit field for representing the sample sizes; it permits defining a constant size for all samples in a track. The second variant permits smaller size fields, to save space when the sizes are varying but small. One of these boxes shall be present; the first version is preferred for maximum compatibility.

NOTE A sample size of zero is not prohibited in general, but it must be valid and defined for the coding system, as defined by the sample entry, that the sample belongs to.

8.7.3.2 Sample size box

8.7.3.2.1 Syntax

```
aligned(8) class SampleSizeBox extends FullBox('stsz', version = 0, 0) {
   unsigned int(32)   sample_size;
   unsigned int(32)   sample_count;
   if (sample_size==0) {
      for (i=1; i <= sample_count; i++) {
      unsigned int(32)   entry_size;
      }
   }
}
```

8.7.3.2.2 Semantics

`version` is an integer that specifies the version of this box

`sample_size` is integer specifying the default sample size. If all the samples are the same size, this field contains that size value. If this field is set to 0, then the samples have different sizes, and those sizes are stored in the sample size table. If this field is not 0, it specifies the constant sample size, and no array follows.

`sample_count` is an integer that gives the number of samples in the track; if sample-size is 0, then it is also the number of entries in the following table.

`entry_size` is an integer specifying the size of a sample, indexed by its number.

8.7.3.3 Compact sample size box

8.7.3.3.1 Syntax

```
aligned(8) class CompactSampleSizeBox
      extends FullBox('stz2', version = 0, 0) {
   unsigned int(24)   reserved = 0;
   unsigned int(8)   field_size;
   unsigned int(32)   sample_count;
   for (i=1; i <= sample_count; i++) {
      unsigned int(field_size)   entry_size;
   }
}
```

8.7.3.3.2 Semantics

`version` is an integer that specifies the version of this box

`field_size` is an integer specifying the size in bits of the entries in the following table; it shall take the value 4, 8 or 16. If the value 4 is used, then each byte contains two values: entry[i]<<4 + entry[i+1]; if the sizes do not fill an integral number of bytes, the last byte is padded with zeros.

`sample_count` is an integer that gives the number of entries in the following table

`entry_size` is an integer specifying the size of a sample, indexed by its number.

8.7.4 Sample to chunk box

8.7.4.1 Definition

Box Type: `'stsc'`
Container: `SampleTableBox`
Mandatory: Yes
Quantity: Exactly one

Samples within the media data are grouped into chunks. Chunks can be of different sizes, and the samples within a chunk can have different sizes. This table can be used to find the chunk that contains a sample, its position, and the associated sample description.

The table is compactly coded. Each entry gives the index of the first chunk of a run of chunks with the same characteristics. By subtracting one entry here from the previous one, it is possible to compute how many chunks are in this run. This can be converted to a sample count by multiplying by the appropriate samples-per-chunk.

8.7.4.2 Syntax

```
aligned(8) class SampleToChunkBox
   extends FullBox('stsc', version = 0, 0) {
   unsigned int(32)   entry_count;
   for (i=1; i <= entry_count; i++) {
      unsigned int(32)   first_chunk;
      unsigned int(32)   samples_per_chunk;
      unsigned int(32)   sample_description_index;
   }
}
```

8.7.4.3 Semantics

`version` is an integer that specifies the version of this box

`entry_count` is an integer that gives the number of entries in the following table

`first_chunk` is an integer that gives the index of the first chunk in this run of chunks that share the same samples-per-chunk and sample-description-index; the index of the first chunk in a track has the value 1 (the `first_chunk` field in the first record of this box has the value 1, identifying that the first sample maps to the first chunk).

`samples_per_chunk` is an integer that gives the number of samples in each of these chunks

`sample_description_index` is an integer that gives the index of the sample entry that describes the samples in this chunk. The index ranges from 1 to the number of sample entries in the `SampleDescriptionBox`

8.7.5 Chunk offset box

8.7.5.1 Definition

Box Type: `'stco'`, `'co64'`
Container: `SampleTableBox`
Mandatory: Yes
Quantity: Exactly one variant shall be present

The chunk offset table gives the index of each chunk into the containing file. There are two variants, permitting the use of 32-bit or 64-bit offsets. The latter is useful when managing very large presentations. At most one of these variants will occur in any single instance of a sample table.

When the referenced data reference entry is not `DataEntryImdaBox` or `DataEntrySeqNumImdaBox`, offsets are file offsets, not the offset into any box within the file (e.g. `MediaDataBox`). This permits referring to media data in files without any box structure. It does also mean that care must be taken when constructing a self-contained ISO file with its structure-data (`MovieBox`) at the front, as the size of the `MovieBox` will affect the chunk offsets to the media data.

When the referenced data reference entry is `DataEntryImdaBox` or `DataEntrySeqNumImdaBox`, offsets are relative to the first byte of the payload of the `IdentifiedMediaDataBox` corresponding to the data reference entry. This permits reordering file-level boxes and receiving a subset of file-level boxes but could require traversing the file-level boxes until the referenced `IdentifiedMediaDataBox` is found.

8.7.5.2 Syntax

```
aligned(8) class ChunkOffsetBox
   extends FullBox('stco', version = 0, 0) {
   unsigned int(32)   entry_count;
   for (i=1; i <= entry_count; i++) {
      unsigned int(32)   chunk_offset;
   }
}
aligned(8) class ChunkLargeOffsetBox
   extends FullBox('co64', version = 0, 0) {
   unsigned int(32)   entry_count;
   for (i=1; i <= entry_count; i++) {
      unsigned int(64)   chunk_offset;
   }
}
```

8.7.5.3 Semantics

`version` is an integer that specifies the version of this box

`entry_count` is an integer that gives the number of entries in the following table

`chunk_offset` is a 32 or 64 bit integer that gives the offset of the start of a chunk. If the referenced data reference entry is `DataEntryImdaBox` or `DataEntrySeqNumImdaBox`, the value of `chunk_offset` is relative to the first byte of the payload of the `IdentifiedMediaDataBox` corresponding to the data reference entry. Otherwise, the value of `chunk_offset` is relative to the start of the containing media file.

8.7.6 Padding bits box

8.7.6.1 Definition

Box Type: `'padb'`
Container: `SampleTableBox`
Mandatory: No
Quantity: Zero or one

In some streams the media samples do not occupy all bits of the bytes given by the sample size, and are padded at the end to a byte boundary. In some cases, it is necessary to record externally the number of padding bits used. This table supplies that information.

8.7.6.2 Syntax

```
aligned(8) class PaddingBitsBox extends FullBox('padb', version = 0, 0) {
   unsigned int(32)   sample_count;
   int i;
   for (i=0; i < floor((sample_count + 1)/2); i++) {
      bit(1)   reserved = 0;
      bit(3)   pad1;
      bit(1)   reserved = 0;
      bit(3)   pad2;
   }
}
```

8.7.6.3 Semantics

`sample_count` counts the number of samples in the track; it should match the count in other tables

`pad1` is a value from 0 to 7, indicating the number of padding bits at the end of sample (i*2)+1.

`pad2` is a value from 0 to 7, indicating the number of padding bits at the end of sample (i*2)+2

8.7.7 Sub-sample information box

8.7.7.1 Definition

Box Type: `'subs'`
Container: `SampleTableBox` or `TrackFragmentBox`
Mandatory: No
Quantity: Zero or more

This box is designed to contain sub-sample information.

A sub-sample is a contiguous range of bytes of a sample. The specific definition of a sub-sample shall be supplied for a given coding system (e.g. for ISO/IEC 14496-10:2014, Advanced Video Coding). In the absence of such a specific definition, this box shall not be applied to samples using that coding system.

If `subsample_count` is 0 for any entry, then those samples have no subsample information and no array follows. The table is sparsely coded; the table identifies which samples have sub-sample structure by recording the difference in sample-number between each entry. The first entry in the table records the sample number of the first sample having sub-sample information.

NOTE It is possible to combine `subsample_priority` and `discardable` such that when `subsample_priority` is smaller than a certain value, `discardable` is set to 1. However, since different systems may use different scales of priority values, separating them is safer, to have a clean solution for discardable sub-samples.

When more than one `SubSampleInformationBox` is present in the same container box, the value of `flags` shall differ in each of these `SubSampleInformationBox`es. The semantics of `flags`, if any, shall be supplied for a given coding system. If `flags` have no semantics for a given coding system, the `flags` shall be 0.

8.7.7.2 Syntax

```
aligned(8) class SubSampleInformationBox
   extends FullBox('subs', version, flags) {
   unsigned int(32) entry_count;
   int i,j;
   for (i=0; i < entry_count; i++) {
      unsigned int(32) sample_delta;
      unsigned int(16) subsample_count;
      if (subsample_count > 0) {
         for (j=0; j < subsample_count; j++) {
            if(version == 1)
            {
               unsigned int(32) subsample_size;
            }
            else
            {
               unsigned int(16) subsample_size;
            }
            unsigned int(8) subsample_priority;
            unsigned int(8) discardable;
            unsigned int(32) codec_specific_parameters;
         }
      }
   }
}
```

8.7.7.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1 in this document)

`entry_count` is an integer that gives the number of entries in the following table.

`sample_delta` is an integer that indicates the sample having sub-sample structure. It is coded as the difference, in decoding order, between the desired sample number, and the sample number indicated in the previous entry. If the current entry is the first entry in the track, the value indicates the sample number of the first sample having sub-sample information, that is, the value is the difference between the sample number and zero (0). If the current entry is the first entry in a track fragment with preceding non-empty track fragments, the value indicates the difference between the sample number of the first sample having sub-sample information and the sample number of the last sample in the previous track fragment. If the current entry is the first entry in a track fragment without any preceding track fragments, the value indicates the sample number of the first sample having sub-sample information, that is, the value is the difference between the sample number and zero (0). This implies that the `sample_delta` for the first entry describing the first sample in the track or in the track fragment is always 1.

`subsample_count` is an integer that specifies the number of sub-sample for the current sample. If there is no sub-sample structure, then this field takes the value 0.

`subsample_size` is an integer that specifies the size, in bytes, of the current sub-sample.

`subsample_priority` is an integer specifying the degradation priority for each sub-sample. Higher values of `subsample_priority`, indicate sub-samples which are important to, and have a greater impact on, the decoded quality.

`discardable` equal to 0 means that the sub-sample is required to decode the current sample, while equal to 1 means the sub-sample is not required to decode the current sample but may be used for enhancements, e.g., the sub-sample consists of supplemental enhancement information (SEI) messages.

`codec_specific_parameters` is defined by the codec in use. If no such definition is available, this field shall be set to 0.

8.7.8 Sample auxiliary information sizes box

8.7.8.1 Definition

Box Type: `'saiz'`
Container: `SampleTableBox` or `TrackFragmentBox`
Mandatory: No
Quantity: Zero or More

Per-sample sample auxiliary information may be stored anywhere in the same file as the sample data itself; for self-contained media files, this is typically in a `MediaDataBox` or a box from a derived specification. It is stored either (a) in multiple chunks, with the number of samples per chunk, as well as the number of chunks, matching the chunking of the primary sample data or (b) in a single chunk for all the samples in a movie sample table (or a movie fragment). The Sample Auxiliary Information for all samples contained within a single chunk (or track run) is stored contiguously (similarly to sample data).

Sample Auxiliary Information, when present, is always stored in the same file as the samples to which it relates as they share the same data reference ('dref') structure. However, this data may be located anywhere within this file, using auxiliary information offsets ('saio') to indicate the location of the data.

Whether sample auxiliary information is permitted or required may be specified by the brands or the coding format in use. The format of the sample auxiliary information is determined by aux_info_type. If aux_info_type and aux_info_type_parameter are omitted then the implied value of aux_info_type is either (a) in the case of transformed content, such as protected content, the scheme_type included in the ProtectionSchemeInfoBox or ScrambleSchemeInfoBox, or otherwise (b) the sample entry type. In the case of tracks containing multiple transformations, aux_info_type and aux_info_type_parameter shall not be omitted. The default value of the aux_info_type_parameter is 0. Some values of aux_info_type may be restricted to be used only with particular track types. A track may have multiple streams of sample auxiliary information of different types. The types are managed according to Annex D.

While aux_info_type determines the format of the auxiliary information, several streams of auxiliary information having the same format may be used when their value of aux_info_type_parameter differs. The semantics of aux_info_type_parameter for a particular aux_info_type value shall be specified along with specifying the semantics of the particular aux_info_type value and the implied auxiliary information format.

This box provides the size of the auxiliary information for each sample. For each instance of this box, there shall be a matching SampleAuxiliaryInformationOffsetsBox with the same values of aux_info_type and aux_info_type_parameter, providing the offset information for this auxiliary information.

NOTE For discussions on the use of sample auxiliary information versus other mechanisms, see Annex B.8.

8.7.8.2 Syntax

```
aligned(8) class SampleAuxiliaryInformationSizesBox
   extends FullBox('saiz', version = 0, flags)
{
   if (flags & 1) {
      unsigned int(32) aux_info_type;
      unsigned int(32) aux_info_type_parameter;
   }
   unsigned int(8) default_sample_info_size;
   unsigned int(32) sample_count;
   if (default_sample_info_size == 0) {
      unsigned int(8) sample_info_size[ sample_count ];
   }
}
```

8.7.8.3 Semantics

aux_info_type is an integer that identifies the type of the sample auxiliary information. At most one occurrence of this box with the same values for aux_info_type and aux_info_type_parameter shall exist in the containing box.

aux_info_type_parameter identifies the “stream” of auxiliary information having the same value of aux_info_type and associated to the same track. The semantics of aux_info_type_parameter are determined by the value of aux_info_type.

default_sample_info_size is an integer specifying the sample auxiliary information size for the case where all the indicated samples have the same sample auxiliary information size. If the size varies then this field shall be zero.

sample_count is an integer that gives the number of samples for which a size is defined. For a SampleAuxiliaryInformationSizesBox appearing in the SampleTableBox this shall be the same as, or less than, the sample_count within the SampleSizeBox or CompactSampleSizeBox. For a SampleAuxiliaryInformationSizesBox appearing in a TrackFragmentBox this shall be the same as, or less than, the sum of the sample_count entries within the TrackRunBoxes of the track fragment. If this is less than the number of samples, then auxiliary information is supplied for the initial samples, and the remaining samples have no associated auxiliary information.

sample_info_size gives the size of the sample auxiliary information in bytes. This may be zero to indicate samples with no associated auxiliary information.

8.7.9 Sample auxiliary information offsets box

8.7.9.1 Definition

Box Type: 'saio'
Container: SampleTableBox or TrackFragmentBox
Mandatory: No
Quantity: Zero or More

For an introduction to sample auxiliary information, see the definition of the SampleAuxiliaryInformationSizesBox.

This box provides the position information for the sample auxiliary information, in a way similar to the chunk offsets for sample data.

8.7.9.2 Syntax

```
aligned(8) class SampleAuxiliaryInformationOffsetsBox
   extends FullBox('saio', version, flags)
{
   if (flags & 1) {
      unsigned int(32) aux_info_type;
      unsigned int(32) aux_info_type_parameter;
   }
   unsigned int(32) entry_count;
   if ( version == 0 ) {
      unsigned int(32) offset[ entry_count ];
   }
   else {
      unsigned int(64) offset[ entry_count ];
   }
}
```

8.7.9.3 Semantics

aux_info_type and aux_info_type_parameter are defined as in the SampleAuxiliaryInformationSizesBox

entry_count gives the number of entries in the following table. For a SampleAuxiliaryInformationOffsetsBox appearing in a Sample Table Box this shall be equal to one or to the value of the entry_count field in the ChunkOffsetBox or ChunkLargeOffsetBox. For a SampleAuxiliaryInformationOffsetsBox appearing in a TrackFragmentBox, this shall be equal to one or to the number of TrackRunBoxes in the TrackFragmentBox.

offset gives the position in the file of the Sample Auxiliary Information for each Chunk or Track Fragment Run. If entry_count is one, then the Sample Auxiliary Information for all Chunks or Runs is contiguous in the file in chunk or run order. When in the SampleTableBox, the offsets are relative to the same base offset as derived for the respective samples through the data_reference_index of the sample entry referenced by the samples. In a TrackFragmentBox, this value is relative to the base offset established by the TrackFragmentHeaderBox in the same track fragment (see 8.8.14).

8.8 Movie fragments

8.8.1 Movie extends box

8.8.1.1 Definition

Box Type: `'mvex'`
Container: `MovieBox`
Mandatory: No
Quantity: Zero or one

This box warns readers that there might be `MovieFragmentBox`es in this file. To know of all samples in the tracks, these `MovieFragmentBox`es must be found and scanned in order, and their information logically added to that found in the Movie Box.

There is a narrative introduction to movie fragments in Annex A.

There are functional equivalences between structures and fields in fragmented and non-fragmented movies, as documented by Table 4.

Table 4 — Equivalences between fragmented and non-fragmented movies

Movie Fragment	***Non-Fragmented Movie***
`MovieExtendsHeaderBox:fragment_duration`	`TrackHeaderBox:duration`

`TrackExtendsBox:default_sample_flags`
`TrackFragmentHeaderBox:default_sample_flags`
`TrackRunBox:(first_sample_flags, sample_flags)`	

`:is_leading`	`SampleDependencyTypeBox:is_leading`
`:sample_depends_on`	`SampleDependencyTypeBox:sample_depends_on`
`:sample_is_depended_on`	`SampleDependencyTypeBox:sample_is_depended_on`
`:sample_has_redundancy`	`SampleDependencyTypeBox:sample_has_redundancy`
`:sample_padding_value`	`PaddingBitsBox:(pad1, pad2)`
`:sample_is_non_sync_sample`	`¬ present(SyncSampleBox:sample_number)`
`:sample_degradation_priority`	`DegradationPriorityBox:priority`
`TrackExtendsBox:default_sample_description_index` `TrackFragmentHeaderBox:sample_description_index`	`SampleToChunkBox:sample_description_index`

`TrackExtendsBox:default_sample_duration`
`TrackFragmentHeaderBox:default_sample_duration`
`TrackRunBox:sample_duration`
	`TimeToSampleBox:sample_delta`

`TrackExtendsBox:default_sample_size`
`TrackFragmentHeaderBox:default_sample_size`
`TrackRunBox:sample_size`
	`SampleSizeBox:(sample_size , entry_size)`
	`CompactSampleSizeBox:entry_size`

`TrackRunBox:sample_composition_time_offset`	`CompositionOffsetBox:sample_offset`

8.8.1.2 Syntax

```
aligned(8) class MovieExtendsBox extends Box('mvex'){
}
```

8.8.2 Movie extends header box

8.8.2.1 Definition

Box Type: `'mehd'`
Container: `MovieExtendsBox`
Mandatory: No
Quantity: Zero or one

The movie extends header is optional, and provides the overall duration, including fragments, of a fragmented movie. If this box is not present, the overall duration must be computed by examining each fragment.

If the duration fields in all tracks are 0, and movie fragments are present, the duration in MovieHeaderBox should be set to indefinite or 0. If movie fragments are present but there is no MediaExtendsHeaderBox and the movie duration is 0, the movie duration should be interpreted as indefinite duration.

8.8.2.2 Syntax

```
aligned(8) class MovieExtendsHeaderBox extends FullBox('mehd', version, 0) {
   if (version==1) {
      unsigned int(64)   fragment_duration;
   } else { // version==0
      unsigned int(32)   fragment_duration;
   }
}
```

8.8.2.3 Semantics

`fragment_duration` is an integer that declares length of the presentation of the whole movie including fragments (in the timescale indicated in the `MovieHeaderBox`). The value of this field corresponds to the duration of the longest track, including movie fragments. If an MP4 file is created in real-time, such as used in live streaming, it is not likely that the `fragment_duration` is known in advance and this box may be omitted.

8.8.3 Track extends box

8.8.3.1 Definition

Box Type: `'trex'`
Container: `MovieExtendsBox`
Mandatory: Yes
Quantity: Exactly one for each track in the `MovieBox`

This sets up default values used by the movie fragments. By setting defaults in this way, space and complexity can be saved in each `TrackFragmentBox`.

The sample flags field in sample fragments (`default_sample_flags` here and in a `TrackFragmentHeaderBox`, and `sample_flags` and `first_sample_flags` in a `TrackRunBox`) is coded as a 32-bit value. It has the following structure:

```
    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_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 `SampleDependencyTypeBox`.

The flag sample_is_non_sync_sample provides the same information as the sync sample table [8.6.2]. When this value is set to 0 for a sample, it is the same as if the sample were not in a movie fragment and marked with an entry in the sync sample table (or, if all samples are sync samples, the sync sample table were absent).

The sample_padding_value is defined as for the PaddingBitsBox. The sample_degradation_priority is defined as for the DegradationPriorityBox.

8.8.3.2 Syntax

```
aligned(8) class TrackExtendsBox extends FullBox('trex', 0, 0){
   unsigned int(32)   track_ID;
   unsigned int(32)   default_sample_description_index;
   unsigned int(32)   default_sample_duration;
   unsigned int(32)   default_sample_size;
   unsigned int(32)   default_sample_flags;
}
```

8.8.3.3 Semantics

track_ID identifies the track; this shall be the track_ID of a track in the MovieBox

default_sample_description_index: indicates the index of the sample entry that describes, by default, the samples in the track fragments

default_sample_duration: indicates the default duration of the samples in the track fragments

default_sample_size: indicates the default size of the samples in the track fragments

default_sample_flags: indicate the default flags values for the samples in the track fragments. (See 8.8.7 for the possible values)

8.8.4 Movie fragment box

8.8.4.1 Definition

Box Type: 'moof'
Container: File
Mandatory: No
Quantity: Zero or more

The movie fragments extend the presentation in time. They provide the information that would previously have been in the MovieBox. The actual samples are in MediaDataBoxes, as usual, if they are in the same file. The data reference index is in the sample description, so it is possible to build incremental presentations where the media data is in files other than the file containing the MovieBox.

The MovieFragmentBox is a top-level box, (i.e. a peer to the MovieBox and MediaDataBoxes). It contains a MovieFragmentHeaderBox, and then one or more TrackFragmentBoxes.

NOTE There is no requirement that any particular movie fragment extend all tracks present in the movie header, and there is no restriction on the location of the media data referred to by the movie fragments. However, derived specifications may make such restrictions.

8.8.4.2 Syntax

```
aligned(8) class MovieFragmentBox extends Box('moof'){
}
```

8.8.5 Movie fragment header box

8.8.5.1 Definition

Box Type: `'mfhd'`
Container: `MovieFragmentBox`
Mandatory: Yes
Quantity: Exactly one

The movie fragment header contains a sequence number, as a safety check. The sequence number usually starts at 1 and increases for each movie fragment in the file, in the order in which they occur. This allows readers to verify integrity of the sequence in environments where undesired re-ordering might occur.

8.8.5.2 Syntax

```
aligned(8) class MovieFragmentHeaderBox
         extends FullBox('mfhd', 0, 0){
   unsigned int(32)   sequence_number;
}
```

8.8.5.3 Semantics

`sequence_number` a number associated with this fragment

8.8.6 Track fragment box

8.8.6.1 Definition

Box Type: `'traf'`
Container: `MovieFragmentBox`
Mandatory: No
Quantity: Zero or more

Within the movie fragment there is a set of track fragments, zero or more per track. The track fragments in turn contain zero or more track runs, each of which documents a contiguous run of samples for that track. Within these structures, many fields are optional and can be defaulted.

It is possible to add 'empty time' to a track using these structures, as well as adding samples. Empty inserts can be used in audio tracks doing silence suppression, for example. These are referred to in this document as ‘empty’ edits (portions of the presentation timeline that map to no media).

8.8.6.2 Syntax

```
aligned(8) class TrackFragmentBox extends Box('traf'){
}
```

8.8.7 Track fragment header box

8.8.7.1 Definition

Box Type: `'tfhd'`
Container: `TrackFragmentBox`
Mandatory: Yes
Quantity: Exactly one

Each movie fragment can add zero or more fragments to each track; and a track fragment can add zero or more contiguous runs of samples. The track fragment header sets up information and defaults used for those runs of samples.

The data origin that the base data offset is relative to and the value of `base_data_offset`, when not present, are inferred as follows:

— If the `base-data-offset-present` flag is equal to 0 and the `default-base-is-moof` flag is equal to 1, `base_data_offset` is inferred to be equal to 0 and is relative to the first byte of the `MovieFragmentBox` containing this box.

— Otherwise, if `base-data-offset-present` flag is equal to 0 and the `default-base-is-moof` flag is equal to 0 and this `TrackFragmentBox` is not the first `TrackFragmentBox` of the same track in the containing `MovieFragmentBox`, `base_data_offset` is inferred to be equal to 1 and is relative to the end of the data defined by the preceding track fragment of the same track.

— Otherwise, if `base-data-offset-present` flag is equal to 0 and the `default-base-is-moof` flag is equal to 0 and the referenced data reference entry is `DataEntryImdaBox` or `DataEntrySeqNumImdaBox`, `base_data_offset` is inferred to be equal to 0 and is relative to the first byte of the payload of the `IdentifiedMediaDataBox` corresponding to the data reference entry.

— Otherwise, if the referenced data reference entry is `DataEntryImdaBox` or `DataEntrySeqNumImdaBox`, the base data offset is relative to the first byte of the payload of the `IdentifiedMediaDataBox` corresponding to the data reference entry.

— Otherwise, the base data offset is relative to the file identified by the referenced data reference entry.

The following flags are defined in the `tf_flags`:

`0x000001 base-data-offset-present`: indicates the presence of the `base-data-offset` field. This provides an explicit anchor for the data offsets in each track run (see below).

`0x000002 sample-description-index-present`: indicates the presence of this field, which over-rides, in this fragment, the default set up in the `TrackExtendsBox`.

```
0x000008   default-sample-duration-present
0x000010   default-sample-size-present
0x000020   default-sample-flags-present
```

`0x010000 duration-is-empty`: this indicates that the duration provided in either `default-sample-duration`, or by the `default-sample-duration` in the `TrackExtendsBox`, is empty, i.e. that there are no samples for this time interval. It is an error to make a presentation that has both edit lists in the `MovieBox`, and empty-duration fragments.

`0x020000 default-base-is-moof`: if `base-data-offset-present` is 1, this flag is ignored. Support for the `default-base-is-moof` flag is required under the `'iso5'` brand, and it shall not be used in brands or compatible brands earlier than `'iso5'`.

NOTE The use of the `default-base-is-moof flag` breaks the compatibility to earlier brands of the file format, because it sets the anchor point for offset calculation differently than earlier. Therefore, the `default-base-is-moof flag` cannot be set when earlier brands are included in the `FileTypeBox`.

Movie-fragment relative addressing is controlled by the values of the base-data-offset-present and default-base-is-moof flags. When both the `base-data-offset-present` flag and the `default-base-is-moof` flag are equal to 0, the value of `data_reference_index` shall be equal in this `TrackFragmentHeaderBox` and in the previous `TrackFragmentHeaderBox` of the same track.

8.8.7.2 Syntax

```
aligned(8) class TrackFragmentHeaderBox
         extends FullBox('tfhd', 0, tf_flags){
   unsigned int(32)   track_ID;
   // all the following are optional fields
   // their presence is indicated by bits in the tf_flags
   unsigned int(64)   base_data_offset;
   unsigned int(32)   sample_description_index;
   unsigned int(32)   default_sample_duration;
   unsigned int(32)   default_sample_size;
   unsigned int(32)   default_sample_flags;
}
```

8.8.7.3 Semantics

`base_data_offset` the base offset to use when calculating data offsets

`sample_description_index`, `default_sample_duration`, `default_sample_size`, `default_sample_flags`:
see 8.8.3.3

8.8.8 Track fragment run box

8.8.8.1 Definition

Box Type: `'trun'`
Container: `TrackFragmentBox`
Mandatory: No
Quantity: Zero or more

Within the `TrackFragmentBox`, there are zero or more `TrackRunBox`es. If the duration-is-empty flag is set in the `tf_flags`, there are no track runs. A track run documents a contiguous set of samples for a track.

The number of optional fields is determined from the number of bits set in the lower byte of the flags, and the size of a record from the bits set in the second byte of the flags. This procedure shall be followed, to allow for new fields to be defined.

If the data-offset is not present, then the data for this run starts immediately after the data of the previous run, or at the base-data-offset defined by the track fragment header if this is the first run in a track fragment, If the data-offset is present, it is relative to the base-data-offset established in the track fragment header.

The following flags are allowed to be set in the `tr_flags`:

0x000001 `data-offset-present`.

0x000004 `first-sample-flags-present`; this overrides the default flags for the first sample only, defined in 8.8.3.1. This makes it possible to record a group of frames where the first is a key and the rest are difference frames, without supplying explicit flags for every sample. If this flag and field are used, `sample-flags-present` shall not be set.

0x000100 `sample-duration-present`: indicates that each sample has its own duration, otherwise the default is used.

0x000200 `sample-size-present`: each sample has its own size, otherwise the default is used.

0x000400 `sample-flags-present`; each sample has its own flags, otherwise the default is used.

0x000800 `sample-composition-time-offsets-present`; each sample has a composition time offset.

The composition offset values in the `CompositionOffsetBox` and in the `TrackRunBox` may be signed or unsigned. The recommendations given in the `CompositionOffsetBox` concerning the use of signed composition offsets also apply here.

8.8.8.2 Syntax

```
aligned(8) class TrackRunBox
         extends FullBox('trun', version, tr_flags) {
   unsigned int(32)    sample_count;
   // the following are optional fields
   signed int(32)    data_offset;
   unsigned int(32)    first_sample_flags;
   // all fields in the following array are optional
   // as indicated by bits set in the tr_flags
   {
      unsigned int(32)    sample_duration;
      unsigned int(32)    sample_size;
      unsigned int(32)    sample_flags
      if (version == 0)
         { unsigned int(32)    sample_composition_time_offset; }
      else
         { signed int(32)      sample_composition_time_offset; }
   }[ sample_count ]
}
```

8.8.8.3 Semantics

`sample_count` the number of samples being added in this run; also the number of rows in the following table (the rows can be empty)

`data_offset` is added to the implicit or explicit `data_offset` established in the track fragment header.

`first_sample_flags` provides a set of flags for the first sample only of this run.

8.8.9 Movie fragment random access box

8.8.9.1 Definition

Box Type: `'mfra'`
Container: File
Mandatory: No
Quantity: Zero or one

The `MovieFragmentRandomAccessBox` provides a table which may assist readers in finding sync samples in a file using movie fragments. It contains a `TrackFragmentRandomAccessBox` for each track for which information is provided (which may not be all tracks). It is usually placed at or near the end of the file; the last box within the `MovieFragmentRandomAccessBox` provides a copy of the length field from the `MovieFragmentRandomAccessBox`. Readers may attempt to find this box by examining the last 32 bits of the file, or scanning backwards from the end of the file for a `MovieFragmentRandomAccessOffsetBox` and using the size information in it, to see if that locates the beginning of a `MovieFragmentRandomAccessBox`.

This box provides only a hint as to where sync samples are; the movie fragments themselves are definitive. It is recommended that readers take care in both locating and using this box as modifications to the file after it was created may render either the pointers, or the declaration of sync samples, incorrect.

8.8.9.2 Syntax

```
aligned(8) class MovieFragmentRandomAccessBox
   extends Box('mfra')
{
}
```

8.8.10 Track fragment random access box

8.8.10.1 Definition

Box Type: `'tfra'`
Container: `MovieFragmentRandomAccessBox`
Mandatory: No
Quantity: Zero or one per track

Each entry contains the location and the presentation time of the sync sample. Not every sync sample in the track needs to be listed in the table.

The absence of this box does not mean that all the samples are sync samples. Random access information in the `'trun'`, `'traf'` and `'trex'` shall be set appropriately regardless of the presence of this box.

8.8.10.2 Syntax

```
aligned(8) class TrackFragmentRandomAccessBox
 extends FullBox('tfra', version, 0) {
   unsigned int(32)   track_ID;
   const unsigned int(26)   reserved = 0;
   unsigned int(2)   length_size_of_traf_num;
   unsigned int(2)   length_size_of_trun_num;
   unsigned int(2)   length_size_of_sample_num;
   unsigned int(32)   number_of_entry;
   for(i=1; i <= number_of_entry; i++){
      if(version==1){
         unsigned int(64)   time;
         unsigned int(64)   moof_offset;
      }else{
         unsigned int(32)   time;
         unsigned int(32)   moof_offset;
      }
      unsigned int((length_size_of_traf_num+1) * 8)   traf_number;
      unsigned int((length_size_of_trun_num+1) * 8)   trun_number;
      unsigned int((length_size_of_sample_num+1) * 8)   sample_delta;
   }
}
```

8.8.10.3 Semantics

`track_ID` is an integer providing the track identifier for which random access information is provided

`length_size_of_traf_num` indicates the length in bytes of the `traf_number` field minus one.

`length_size_of_trun_num` indicates the length in bytes of the `trun_number` field minus one.

`length_size_of_sample_num` indicates the length in bytes of the `sample_number` field minus one.

`number_of_entry` is an integer that gives the number of the entries for this track. If this value is zero, it indicates that every sample is a sync sample and no table entry follows.

`time` is a 32 or 64 bit integer that indicates the presentation time of the sync sample in units defined in the `MediaHeaderBox` of the associated track.

NOTE Presentation times are usually expressed in movie timescale, except for the specific case of `TrackFragmentRandomAccessBox` where it is expressed in media timescale.

`moof_offset` is a 32 or 64 bits integer that gives the offset of the `'moof'` used in this entry. Offset is the byte-offset between the beginning of the file and the beginning of the `'moof'`.

`traf_number` indicates the `'traf'` number that contains the sync sample. The number ranges from 1 (the first `'traf'` is numbered 1) in each `'moof'`.

`trun_number` indicates the `'trun'` number that contains the sync sample. The number ranges from 1 in each `'traf'`.

`sample_delta` indicates the sample number of the sync sample. It is coded as one plus the desired sample number minus the sample number of the first sample in the `TrackRunBox`.

8.8.11 Movie fragment random access offset box

8.8.11.1 Definition

Box Type: `'mfro'`
Container: `MovieFragmentRandomAccessBox`
Mandatory: Yes
Quantity: Exactly one

The `MovieFragmentRandomAccessOffsetBox` provides a copy of the size field from the enclosing `MovieFragmentRandomAccessBox`. It is placed last within that box, so that the size field is also last in the enclosing `MovieFragmentRandomAccessBox`. When the `MovieFragmentRandomAccessBox` is also last in the file this permits its easy location. The size field here shall be correct. However, neither the presence of the `MovieFragmentRandomAccessBox`, nor its placement last in the file, are assured.

8.8.11.2 Syntax

```
aligned(8) class MovieFragmentRandomAccessOffsetBox
 extends FullBox('mfro', version, 0) {
   unsigned int(32)    parent_size;
}
```

8.8.11.3 Semantics

`parent_size` is an integer that gives the number of bytes of the enclosing `MovieFragmentRandomAccessBox` box. This field is placed last in the enclosing box to assist readers scanning from the end of the file in finding the `MovieFragmentRandomAccessBox`.

8.8.12 Track fragment decode time box

8.8.12.1 Definition

Box Type: `'tfdt'`
Container: `TrackFragmentBox`
Mandatory: No
Quantity: Zero or one

The `TrackFragmentBaseMediaDecodeTimeBox` provides the absolute decoding timestamp, measured on the decoding timeline, of the first sample in decoding order in the track fragment. This can be useful, for example, when performing random access in a file; it is not necessary to sum the sample durations of all preceding samples in previous fragments to find this value.

The `TrackFragmentBaseMediaDecodeTimeBox`, if present, shall be positioned after the `TrackFragmentHeaderBox` and before the first `TrackRunBox`.

NOTE 1 The decoding timeline is a media timeline, established before any explicit or implied mapping of composition time to presentation time, for example by an edit list or similar structure. See.6.4

If the time expressed in the `TrackFragmentBaseMediaDecodeTimeBox` exceeds the sum of the sample durations of the samples in the preceding movie and movie fragments, then the duration of the last sample preceding this track fragment is extended such that the sum now equals the time given in this box. In this way, it is possible to generate a fragment containing a sample when the time of the next sample is not yet known, by assigning it a small or even zero sample duration, that is then overriden by the time expressed in this box in the following fragment.

If no samples were present in the preceding movie and movie fragments for this track, the time expressed in the `TrackFragmentBaseMediaDecodeTimeBox` defines the decoding timestamp of the first sample in this track.

Players may choose to skip over an initial empty media range in tracks where the first decoding timestamp is defined by a `TrackFragmentBaseMediaDecodeTimeBox` with non-zero time.

In particular, an empty track fragment (with no samples, but with a track fragment decode time box) may be used to establish the duration of the last sample.

NOTE 2 If fragments are delivered out-of-order, file readers might need additional information to determine if the duration of the last sample of the last fragment needs to be extended using the track decode time of the next fragment to come, or if it should wait until all fragments have been received (also indicated by out-of-band means) to do that operation. Derived specifications are permitted to require specific handling of the `sequence_number` of the `MovieFragmentHeaderBox`, for example (e.g. that sequence numbers start at and increment by 1).

8.8.12.2 Syntax

```
aligned(8) class TrackFragmentBaseMediaDecodeTimeBox
   extends FullBox('tfdt', version, 0) {
   if (version==1) {
      unsigned int(64) baseMediaDecodeTime;
   } else { // version==0
      unsigned int(32) baseMediaDecodeTime;
   }
}
```

8.8.12.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1 in this document).

`baseMediaDecodeTime` is an integer equal to the sum of the decode durations of all earlier samples in the media, expressed in the media's timescale. It does not include the samples added in the enclosing track fragment.

8.8.13 Level assignment box

8.8.13.1 Definition

Box Type: `'leva'`
Container: `MovieExtendsBox`
Mandatory: No
Quantity: Zero or one

Levels specify subsets of the file. Samples mapped to level n may depend on any samples of levels m, where m <= n, and shall not depend on any samples of levels p, where p > n. For example, levels can be specified according to temporal level (e.g., temporal_id of SVC or MVC).

Levels cannot be specified for the initial movie. When the `LevelAssignmentBox` is present, it applies to all movie fragments subsequent to the initial movie.

For the context of the `LevelAssignmentBox`, a fraction is defined to consist of one or more `MovieFragmentBox`es and the associated `MediaDataBox`es, possibly including only an initial part of the last `MediaDataBox`. Within a fraction, data for each level shall appear contiguously. Data for levels within a fraction shall appear in increasing order of level value. All data in a fraction shall be assigned to levels.

NOTE In the context of DASH (ISO/IEC 23009-1[21]), each subsegment indexed within a `SubsegmentIndexBox` is a fraction.

The `LevelAssignmentBox` provides a mapping from features, such as scalability layers, to levels. A feature can be specified through a track, a sub-track within a track, or a sample grouping of a track.

When `padding_flag` is equal to 1 this indicates that a conforming fraction can be formed by concatenating any positive integer number of levels within a fraction and padding the last `MediaDataBox` by zero bytes up to the full size that is indicated in the header of the last `MediaDataBox`. The use of `padding_flag` is deprecated.

8.8.13.2 Syntax

```
aligned(8) class LevelAssignmentBox extends FullBox('leva', 0, 0)
{
   unsigned int(8)   level_count;
   for (j=1; j <= level_count; j++) {
      unsigned int(32)   track_ID;
      unsigned int(1)   padding_flag;
      unsigned int(7)   assignment_type;
      if (assignment_type == 0) {
         unsigned int(32)   grouping_type;
      }
      else if (assignment_type == 1) {
         unsigned int(32)   grouping_type;
         unsigned int(32)   grouping_type_parameter;
      }
      else if (assignment_type == 2) {}
         // no further syntax elements needed
      else if (assignment_type == 3) {}
         // no further syntax elements needed
      else if (assignment_type == 4) {
         unsigned int(32) sub_track_ID;
      }
      // other assignment_type values are reserved
   }
}
```

8.8.13.3 Semantics

`level_count` specifies the number of levels each fraction is grouped into. `level_count` shall be greater than or equal to 2.

`track_ID` for loop entry j specifies the track identifier of the track assigned to level j.

`padding_flag` equal to 1 indicates that a conforming fraction can be formed by concatenating any positive integer number of levels within a fraction and padding the last `MediaDataBox` by zero bytes up to the full size that is indicated in the header of the last `MediaDataBox`. When `padding_flag` is equal to 0 this is not assured.

`assignment_type` indicates the mechanism used to specify the assignment to a level. `assignment_type` values greater than 4 are reserved, while the semantics for the other values are specified as follows. The sequence of assignment_types is restricted to be a set of zero or more of type 2 or 3, followed by zero or more of exactly one type.

— 0: sample groups are used to specify levels, i.e., samples mapped to different sample group description indexes of a particular sample grouping lie in different levels within the identified track; other tracks are not affected and shall have all their data in precisely one level;

— 1: as for assignment_type 0 except assignment is by a parameterized sample group;

— 2, 3: level assignment is by track (see the `SubsegmentIndexBox` for the difference in processing of these levels)

— 4: the respective level contains the samples for a sub-track. The sub-tracks are specified through the `SubTrackBox`; other tracks are not affected and shall have all their data in precisely one level;

`grouping_type` and `grouping_type_parameter`, if present, specify the sample grouping used to map sample group description entries in the `SampleGroupDescriptionBox` to levels. Level n contains the samples that are mapped to the `SampleGroupDescriptionEntry` having index n in the SampleGroupDescriptionBox having the same values of grouping_type and grouping_type_parameter, if present, as those provided in this box.

sub_track_ID specifies that the sub-track identified by sub_track_ID within loop entry j is mapped to level j.

8.8.14 Sample auxiliary information in movie fragments

When sample auxiliary information (8.7.8 and 8.7.9) is present in the MovieFragmentBox, the offsets in the SampleAuxiliaryInformationOffsetsBox are treated the same as the data_offset in the TrackRunBox, that is, they are relative to any base data offset established for that track fragment.

If only one offset is provided, then the Sample Auxiliary Information for all the track runs in the fragment is stored contiguously, otherwise exactly one offset shall be provided for each track run.

If the field default_sample_info_size is non-zero in one of these boxes, then the size of the auxiliary information is constant for the identified samples.

In addition, if:

— this box is present in the MovieBox,

— and default_sample_info_size is non-zero in the box in the MovieBox,

— and the SampleAuxiliaryInformationSizesBox is absent in a movie fragment,

then the auxiliary information has this same constant size for every sample in the movie fragment also; it is then not necessary to repeat the box in the movie fragment.

8.8.15 Track Extension Properties box

8.8.15.1 Definition

Box Type: 'trep'
Container: MovieExtendsBox
Mandatory: No
Quantity: Zero or more. (Zero or one per track)

This box can be used to document or summarize characteristics of the track in the subsequent movie fragments. It may contain any number of child boxes.

8.8.15.2 Syntax

```
class TrackExtensionPropertiesBox extends FullBox('trep', 0, 0) {
   unsigned int(32) track_ID;
   // Any number of boxes may follow
}
```

8.8.15.3 Semantics

track_ID indicates the track for which the track extension properties are provided in this box.

8.8.16 Alternative startup sequence properties box

8.8.16.1 Definition

Box Type: 'assp'
Container: TrackExtensionPropertiesBox
Mandatory: No
Quantity: Zero or one

This box indicates the properties of alternative startup sequence sample groups in the subsequent track fragments of the track indicated in the containing `TrackExtensionPropertiesBox`.

Version 0 of the `AlternativeStartupSequencePropertiesBox` shall be used if version 0 of the `SampleToGroupBox` is used for the alternative startup sequence sample grouping. Version 1 of the `AlternativeStartupSequencePropertiesBox` shall be used if version 1 of the `SampleToGroupBox` is used for the alternative startup sequence sample grouping.

8.8.16.2 Syntax

```
class AlternativeStartupSequencePropertiesBox extends FullBox('assp', version, 0) {
   if (version == 0) {
      signed int(32)       min_initial_alt_startup_offset;
   }
   else if (version == 1) {
      unsigned int(32)    num_entries;
      for (j=1; j <= num_entries; j++) {
         unsigned int(32)    grouping_type_parameter;
         signed int(32)       min_initial_alt_startup_offset;
      }
   }
}
```

8.8.16.3 Semantics

`min_initial_alt_startup_offset`: No value of sample_offset[1] of the referred sample group description entries of the alternative startup sequence sample grouping shall be smaller than min_initial_alt_startup_offset. In version 0 of this box, the alternative startup sequence sample grouping using version 0 of the Sample to Group box is referred to. In version 1 of this box, the alternative startup sequence sample grouping using version 1 of the `SampleToGroupBox` is referred to as further constrained by `grouping_type_parameter`.

`num_entries` indicates the number of alternative startup sequence sample groupings documented in this box.

`grouping_type_parameter` indicates which one of the alternative sample groupings this loop entry applies to.

8.8.17 Metadata and user data in movie fragments

When `MetaBox`es occur in `MovieFragmentBox`es or `TrackFragmentBox`es, the following applies. The file shall have been fragmented such that any metadata needed in the movie or track fragment is formed from the union of the metadata in the `MovieBox` and the fragment, not considering or using metadata in any other fragment. Metadata in a movie or track fragment is logically 'arriving late' but is valid for the entire track. When a file is de-fragmented, the metadata in the movie or track fragments must be merged into the movie or track boxes, respectively. This process allows for 'just in time' delivery of support resources, and bandwidth management, while preserving the essentially atemporal nature of untimed metadata. If metadata truly changes over time, a timed metadata track may be needed.

If, during this merge, there are either (a) metadata items with the same `item_ID` or (b) user-data items with the same type, then the following applies:

a) all occurrences of the data (user-data box or metadata item) must be 'true' for the entire movie including all fragments;

b) the occurrences in higher-numbered movie fragments ('later' occurrences) may be more accurate or 'preferred';

c) in particular, data in an empty initial `MovieBox` may be only estimates or 'not to exceed' values, and data in a final otherwise empty movie fragment may be the 'final' or most accurate values.

Consequently, for `MetaBox`, the redefinition of an item with the same `item_ID` in a subsequent `MetaBox` is equivalent to an item replacement, the new item applying for the entire track or file. For `UserDataBox`, there may be multiple occurrences of a user data of a given type (for example CopyrightBox with different languages), and these occurrences may be delivered in different movie fragments or in the initial movie. Readers should be careful when attempting at removing duplicates of such boxes in a defragmentation process, since checking the type of the user data for removal might not be sufficient; the language or other fields specific to the type of user-data may need inspection.

8.9 Sample group structures

8.9.1 Overview

This clause specifies a generic mechanism for representing a partition of the samples in a track. A *sample grouping* is an assignment of each sample in a track to be a member of one *sample group*, based on a grouping criterion. A sample group in a sample grouping is not limited to being contiguous samples and may contain non-adjacent samples. As there may be more than one sample grouping for the samples in a track, each sample grouping has a type field to indicate the type of grouping. For example, a file might contain two sample groupings for the same track: one based on an assignment of sample to layers and another to sub-sequences.

Sample groupings are represented by two linked data structures: (1) a SampleToGroupBox represents the assignment of samples to sample groups; (2) a SampleGroupDescriptionBox contains a *sample group entry* for each sample group describing the properties of the group. There may be multiple instances of the SampleToGroupBox and SampleGroupDescriptionBoxes based on different grouping criteria. These are distinguished by a type field used to indicate the type of grouping.

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 SampleToGroupBox occurs once for each stream, it is now both compact, and informs the reader about each stream separately.

One example of using these tables is to represent the assignments of samples to *layers*. In this case each sample group represents one layer, with an instance of the SampleToGroupBox describing which layer a sample belongs to.

In general it is not required that a sample-to group mapping mark every sample for which the associated sample group description applies, only that the mapping be correct for samples so mapped; however, this general principle may be over-ridden by specific sample groups.

NOTE There might not be a SampleToGroupBox of a given grouping_type corresponding to a SampleGroupDescriptionBox with the same grouping_type because references to the SampleGroupDescriptionBox (in particular to its entries) might be provided by specific constructs in derived specifications.

8.9.2 Sample to group box

8.9.2.1 Definition

Box Type: 'sbgp'
Container: SampleTableBox or TrackFragmentBox
Mandatory: No
Quantity: Zero or more.

This table can be used to find the group that a sample belongs to and the associated description of that sample group. The table is compactly coded with each entry giving the index of the first sample of a run of samples with the same sample group descriptor. The sample group description ID is an index that refers to a SampleGroupDescriptionBox, which contains entries describing the characteristics of each sample group.

There may be multiple instances of this box if there is more than one sample grouping for the samples in a track or track fragment. Each instance of the `SampleToGroup` box has a type that distinguishes different sample groupings. Within a track, whether declared in the `SampleTableBox` or in `TrackFragmentBox`, there shall be at most one instance of this box with a particular `grouping_type`, and, if present, a `grouping_type_parameter`. The associated `SampleGroupDescriptionBox` shall indicate the same value for the `grouping_type`. When there are multiple `SampleToGroupBox`es with a particular value of `grouping_type` in a container box, the version of all the `SampleToGroupBox`es shall be 1. When the version of a `SampleToGroupBox` is 0, there shall be only one occurrence of `SampleToGroupBox` with this `grouping_type` in a container box.

Version 1 of this box should only be used if a `grouping_type_parameter` is needed. When the `grouping_type_parameter` is not explicitly defined in this standard, its semantics may be overridden by derived specifications.

For a `SampleGroupDescriptionBox` with a given `grouping_type`, there may be more than one `SampleToGroupBox` with the same `grouping_type` if and only if each `SampleToGroupBox` has a different value of `grouping_type_parameter`; there may also be no `SampleToGroupBox` with the given `grouping_type` if no samples are mapped to a description of that `grouping_type`, or if all samples are mapped to the default entry identified by the `SampleGroupDescriptionBox`.

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;
   }
}
```

8.9.2.3 Semantics

`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

`entry_count` is an integer that gives the number of entries in the following table.

`sample_count` is an integer that gives the number of consecutive samples with the same sample group descriptor. It is an error for the total in this box to be greater than the `sample_count` documented elsewhere, and the reader behaviour would then be undefined. If the sum of the sample count in this box is less than the total sample count, or there is no `SampleToGroupBox` that applies to some samples (e.g. it is absent from a track fragment), then those samples are associated with the group identified by the `default_group_description_index` in the `SampleGroupDescriptionBox`, if any, or else with no group.

`group_description_index` is an integer that gives the index of the sample group entry which describes the samples in this group. The index ranges from 1 to the number of sample group entries in the `SampleGroupDescriptionBox`, or takes the value 0 to indicate that this sample is a member of no group of this type.

8.9.3 Sample group description box

8.9.3.1 Definition

Box Type: `'sgpd'`
Container: `SampleTableBox` or `TrackFragmentBox`
Mandatory: No
Quantity: Zero or more, with exactly one for each `grouping_type` in a `SampleToGroupBox`.

This description table gives information about the characteristics of sample groups. The descriptive information is any other information needed to define or characterize the sample group. The syntax of the sample group description entry used is determined by both the grouping_type and the media handler type.

There may be multiple instances of this box if there is more than one sample grouping for the samples in a track. Each instance of the `SampleGroupDescriptionBox` has a type that distinguishes different sample groupings. There shall be at most one instance of this box with a particular `grouping_type` in a track (i.e. defined in a `SampleTableBox` or `TrackFragmentBox`).

The flags field of the `SampleGroupDescriptionBox` shall be zero when the box is in a `TrackFragmentBox`. When the box is in a `SampleTableBox`, either or both of the two lowest bits may be set:

— `static_group_description`, with value 1: when set to 1, this flag indicates that there are no `SampleGroupDescriptionBoxes` of this `grouping_type` in any `TrackFragmentBox` of this track.

— `static_mapping`, with value 2: when set to 1, this flag indicates that there are no `SampleToGroupBox`es of this `grouping_type` in this track (in neither the `SampleTableBox` nor any `TrackFragmentBox` of this track); all samples therefore map to the default.

NOTE 1 the `static_mapping` flag is only useful when `default_group_description_index` is non-zero, since the default value of `default_group_description_index` is 0, indicating no mapping.

These flags may be used in combination with the version of the `SampleGroupDescriptionBox` to signal various possibilities.

— `static_group_description` without `static_mapping`:

the sample group definitions are only in the `MovieBox`, but samples can map to any of them.

— `static_mapping` without `static_group_description`:

everything in a fragment maps to at most one group; there may be new `SampleGroupDescriptionBoxes` of this type in fragments; depending on their version, the `SampleGroupDescriptionBoxes` can identify a default sample group, or that samples are unmapped.

— both `static_group_description` and `static_mapping`:

every sample maps to the default indicated in the `SampleGroupDescriptionBox` in the `MovieBox`; that `SampleGroupDescriptionBox` can indicate a default sample group or indicate that all samples are unmapped, depending on its version.

The information is stored in the `SampleGroupDescriptionBox` after the entry-count. An abstract entry type is defined and sample groupings shall define derived types to represent the description of each sample group.

NOTE 2 In version 0 of the entries the base classes for sample group description entries are neither boxes nor have a size that is signalled. For this reason, use of version 0 entries is deprecated. When defining derived classes, ensure either that they have a fixed size, or that the size is explicitly indicated with a length field. An implied size (e.g. achieved by parsing the data) is not recommended as this makes scanning the array difficult.

8.9.3.2 Syntax

```
// Sequence Entry
abstract class SampleGroupDescriptionEntry (unsigned int(32) grouping_type)
{
}
abstract class VisualSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class AudioSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class HintSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class SubtitleSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class TextSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class HapticSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
abstract class VolumetricVisualSampleGroupEntry (unsigned int(32) grouping_type) extends
SampleGroupDescriptionEntry (grouping_type)
{
}
aligned(8) class SampleGroupDescriptionBox ()
   extends FullBox('sgpd', version, flags){
   unsigned int(32) grouping_type;
   if (version>=1) { unsigned int(32) default_length; }
   if (version>=2) {
      unsigned int(32) default_group_description_index;
   }
   unsigned int(32) entry_count;
   int i;
   for (i = 1 ; i <= entry_count ; i++){
      if (version>=1) {
         if (default_length==0) {
            unsigned int(32) description_length;
         }
      }
      SampleGroupDescriptionEntry (grouping_type);
      // an instance of a class derived from SampleGroupDescriptionEntry
      // that is appropriate and permitted for the media type
   }
}
```

8.9.3.3 Semantics

`version` is an integer that specifies the version of this box.

`grouping_type` is an integer that identifies the `SampleToGroupBox` that is associated with this sample group description.

`default_group_description_index`: specifies the index of the sample group description entry which applies to all samples in the track for which no sample to group mapping is provided through a `SampleToGroupBox`. The default value of this field is zero (indicating that the samples are mapped to no group description of this type).

`entry_count` is an integer that gives the number of entries in the following table.

default_length indicates the length of every group entry (if the length is constant), or zero (0) if it is variable

description_length indicates the length of an individual group entry, in the case it varies from entry to entry and default_length is therefore 0

NOTE The field default_group_description_index used to be called default_sample_description_index in previous editions of this document.

8.9.4 Representation of group structures in movie fragments

Support for sample group structures within movie fragments is provided by the use of the SampleToGroupBox with the container for this box being the TrackFragmentBox. The definition, syntax and semantics of this Box is as specified in subclause 8.9.2.

The SampleToGroupBox can be used to find the group that a sample in a track fragment belongs to and the associated description of that sample group. The table is compactly coded with each entry giving the index of the first sample of a run of samples with the same sample group descriptor. The sample group description ID is an index that refers to a SampleGroupDescriptionBox, which contains entries describing the characteristics of each sample group and present in the SampleTableBox.

There may be multiple instances of the SampleToGroupBox if there is more the one sample grouping for the samples in a track fragment. Each instance of the SampleToGroupBox has a grouping_type, and possibly grouping_type_parameter, that distinguishes different sample groupings. The associated SampleGroupDescriptionBox shall indicate the same value for the grouping_type.

The total number of samples represented in any SampleToGroupBox in the track fragment shall match the total number of samples in all the track fragment runs. Each SampleToGroupBox documents a different grouping of the same samples.

Zero or more SampleGroupDescriptionBoxes may also be present in a TrackFragmentBox. These definitions are additional to the definitions provided in the SampleTableBox of the track. Group definitions within a movie fragment can only be referenced and used from within that same movie fragment.

Within the SampleToGroupBox in that movie fragment, the group description indexes for groups defined within the same fragment start at 0x10001, i.e. the index value 1, with the value 1 in the top 16 bits. This means there must be fewer than 65536 group definitions for this track and grouping_type in the SampleTableBox of the track.

When changing the size of movie fragments, or removing them, these fragment-local group definitions will need to be merged into the definitions in the MovieBox, or into the new movie fragments, and the index numbers in the SampleToGroupBox(es) adjusted accordingly. It is recommended that, in this process, identical (and hence duplicate) definitions not be made in any SampleGroupDescriptionBox, but that duplicates be merged and the indexes adjusted accordingly.

When a SampleGroupDescriptionBox "A", declaring a default_group_description_index, is present in a TrackFragmentBox, that default_group_description_index shall indicate either:

— 0 or 0x10000, if no default sample group description is defined for that track fragment

— the index of a sample group description entry in a SampleGroupDescriptionBox of the same grouping_type declared in the SampleTableBox of the track

— the index incremented by 0x10000 of the entry of that SampleGroupDescriptionBox "A".

The index value for the first entry in a SampleGroupDescriptionBox is 1.

For any track fragment, the default sample group description entry for a given grouping_type is the one indicated by the default_group_description_index of the SampleGroupDescriptionBox of the same grouping_type declared in that track fragment, if any; otherwise by default_group_description_index of the SampleGroupDescriptionBox of the same grouping_type declared in the SampleTableBox of the track, if any; otherwise, no default entry is used for that grouping_type.

The default sample group identified in a SampleGroupDescriptionBox in the SampleTableBox shall identify a group within that box, i.e. the index value of the default is less than 0x10000.

8.9.5 Compact sample to group box

8.9.5.1 Definition

Box Type: 'csgp'
Container: SampleTableBox or TrackFragmentBox
Mandatory: No
Quantity: Zero or more.

The compact sample to group box provides a more compact way to represent the mapping from sample to group, especially in the cases where there are repeating patterns, and when there are few sample groups of a particular type.

The design uses a vector of concatenated *patterns* each of which is used once by a mapping array, which associates runs of samples with repeats of that pattern. This is illustrated by the following example. In the following, each letter represents a different sample group description index value (possibly 0).

If a track has the following associations, starting from the first sample:

```
   a b c b  a b c b  a b c x x  a b c b  a b d b
```

those associations might be represented by the following:

```
   1. pattern_length=4; sample_count=11;
   2. pattern_length=1; sample_count=2;
   3. pattern_length=4; sample_count=6;
   4. pattern_length=2; sample_count=2;

   pattern=[
      a b c b      // pattern 1 of length 4
      x            // pattern 2 of length 1
      a b c b      // pattern 3 of length 4
      d b         // pattern 4 of length 2
   ]               // the pattern_length is thus 4+1+4+2=11
```

When sample_count[i] is equal to pattern_length[i], the pattern is not repeated.

When sample_count[i] is greater than pattern_length[i], the sample_group_description_index values of the i-th pattern are used repeatedly to map the sample_count[i] values. It is not necessarily the case that sample_count[i] is a multiple of pattern_length[i]; the cycling may terminate in the middle of the pattern.

When the total of the sample_count[i] values for all values of i in the range of 1 to pattern_count, inclusive, is less than the total sample count, the reader should associate the samples that have no explicit group association with the default group defined in the SampleDescriptionGroupBox, if any, or else with no group.

It is an error for the total of the sample_count[i] values to be greater than the total count of actual samples described by the encompassing TrackBox or TrackFragmentBox, and the reader behaviour would then be undefined.

The 24-bit flag field is defined as follows:

```
   unsigned int(16) reserved = 0;
   unsigned int(1) index_msb_indicates_fragment_local_description;
   unsigned int(1) grouping_type_parameter_present;
   unsigned int(2) pattern_size_code;
   unsigned int(2) count_size_code;
   unsigned int(2) index_size_code;
```

8.9.5.2 Syntax

```
unsigned int(8) function f(unsigned int(2) index) {
   switch(index) {
      case 0: return 4;
      case 1: return 8;
      case 2: return 16;
      case 3: return 32;
}
aligned(8) class CompactSampleToGroupBox
   extends FullBox('csgp', version, flags)
{
   unsigned int(32) grouping_type;

   if (grouping_type_parameter_present == 1) {
      unsigned int(32) grouping_type_parameter;
   }
   unsigned int(32) pattern_count;
   totalPatternLength = 0;
   for (i=1; i <= pattern_count; i++) {
      unsigned int(f(pattern_size_code)) pattern_length[i];
      unsigned int(f(count_size_code)) sample_count[i];
   }
   for (j=1; j <= pattern_count; j++) {
      for (k=1; k <= pattern_length[j]; k++) {
         unsigned int(f(index_size_code))
                    sample_group_description_index[j][k];
         // whose msb might indicate fragment_local or global
      }
   }
}
```

8.9.5.3 Semantics

`version` is an integer that specifies the version of this box, currently 0.

`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 either the `'csgp'` or `'sbgp'` 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.

`index_msb_indicates_fragment_local_description` is a flag that shall be zero when this box appears inside a `TrackBox` but may be 0 or 1 when this box appears inside a `TrackFragmentBox`. When it is 1, it indicates that the most significant bit (MSB) of every `sample_group_description_index` does not form part of the index number but instead indicates which `SampleGroupDescriptionBox` the group description is to be found in: if the MSB is 0, the index identifies a group description from the `TrackBox`'s `SampleGroupDescriptionBox`; if the MSB is 1, the index identifies a group description from the `TrackFragmentBox`'s `SampleGroupDescriptionBox`.

`field_size_code`, `pattern_size_code`, `index_size_code` are integers specifying the size in bits of the entries in the array of the `pattern_length`, `sample_count` and `sample_group_description_index` values respectively; the matching code values map to the sizes: code 0 indicates a 4-bit size, code 1 an 8-bit size, code 2 a 16-bit size, and code 3 a 32-bit size. If the field size 4 is used for the pattern_size, then it shall also be used for the count_size (to maintain byte alignment), though this is probably rarely useful; if it is used for `sample_group_description_index`, then each byte contains two values: `entry[i]<<4 + entry[i+1]`; if the sizes do not fill an integral number of bytes, the last byte is padded with zeros.

`pattern_count` indicates the length of the associated pattern in the pattern array that follows it. The sum of the included `sample_count` values indicates the number of mapped samples.

`pattern_length[i]` corresponds to a pattern within the second array of `sample_group_description_index[j]` values. Each instance of `pattern_length[i]` shall be greater than 0.

`sample_count[i]` specifies the number of samples that use the i-th pattern. `sample_count[i]` shall be greater than zero, and `sample_count[i]` shall be greater than or equal to `pattern_length[i]`.

`sample_group_description_index[j][k]` is an integer that gives the index of the sample group entry which describes the samples in this group. The index ranges from 1 to the number of sample group entries in the `SampleGroupDescriptionBox`, inclusive, or takes the value 0 to indicate that this sample is a member of no group of this type.

8.10 User data

8.10.1 User data box

8.10.1.1 Definition

Box Type: `'udta'`
Container: `MovieBox`, `TrackBox`, `MovieFragmentBox` or `TrackFragmentBox`
Mandatory: No
Quantity: Zero or one

This box contains objects that declare user information about the containing box and its data (presentation or track).

The User Data Box is a container box for informative user-data. This user data is formatted as a set of boxes with more specific box types, which declare more precisely their content. The contained boxes are normal boxes, using a defined, registered, or UUID extension box type.

The handling of user-data in movie fragments is described in 8.8.17.

8.10.1.2 Syntax

```
aligned(8) class UserDataBox extends Box('udta') {
}
```

8.10.2 Copyright box

8.10.2.1 Definition

Box Type: `'cprt'`
Container: `UserDataBox`
Mandatory: No
Quantity: Zero or more

The `CopyrightBox` contains a copyright declaration which applies to the entire presentation, when contained within the `MovieBox`, or, when contained in a track, to that entire track. There may be multiple copyright boxes using different language codes.

8.10.2.2 Syntax

```
aligned(8) class CopyrightBox
   extends FullBox('cprt', version = 0, 0) {
   const bit(1) pad = 0;
   unsigned int(5)[3] language; // ISO-639-2/T language code
   utfstring notice;
}
```

8.10.2.3 Semantics

`language` declares the language code for the following text, in the form of a packed three-character code from ISO 639-2. Each character is packed as the difference between its ASCII value and 0x60. The code is confined to being three lower-case letters, so these values are strictly positive.

`notice` gives a copyright notice.

8.10.3 Track selection box

8.10.3.1 Overview

A typical presentation stored in a file contains one alternate group per media type: one for video, one for audio, etc. Such a file may include several video tracks, although, at any point in time, only one of them should be played or streamed. This is achieved by assigning all video tracks to the same alternate group. (See subclause 8.3.2 for the definition of alternate groups.)

All tracks in an alternate group are candidates for media selection, but it may not make sense to switch between some of those tracks during a session. One may for instance allow switching between video tracks at different bitrates and keep frame size but not allow switching between tracks of different frame size. In the same manner it may be desirable to enable selection – but not switching – between tracks of different video codecs or different audio languages.

The distinction between tracks for *selection* and *switching* is addressed by assigning tracks to switch groups in addition to alternate groups. One alternate group may contain one or more switch groups. All tracks in an alternate group are candidates for media selection, while tracks in a switch group are also available for switching during a session. Different switch groups represent different operation points, such as different frame size, high/low quality, etc.

For the case of non-scalable bitstreams, several tracks may be included in a switch group. The same also applies to non-layered scalable bitstreams, such as traditional AVC streams.

By labelling tracks with attributes it is possible to characterize them. Each track can be labelled with a list of attributes which can be used to describe tracks in a particular switch group or differentiate tracks that belong to different switch groups.

8.10.3.2 Definition

Box Type: `'tsel'`
Container: `UserDataBox` of the corresponding `TrackBox`
Mandatory: No
Quantity: Zero or One

The track selection box is contained in the user data box of the track it modifies.

8.10.3.3 Syntax

```
aligned(8) class TrackSelectionBox
   extends FullBox('tsel', version = 0, 0) {
   template int(32) switch_group = 0;
   unsigned int(32) attribute_list[];      // to end of the box
}
```

8.10.3.4 Semantics

`switch_group` is an integer that specifies a group or collection of tracks. If this field is 0 (default value) or if the `TrackSelectionBox` is absent there is no information on whether the track can be used for switching during playing or streaming. If this integer is not 0 it shall be the same for tracks that can be used for switching between each other. Tracks that belong to the same switch group shall belong to the same alternate group. A switch group may have only one member.

`attribute_list` is a list, to the end of the box, of attributes. The attributes in this list should be used as descriptions of tracks or differentiation criteria for tracks in the same alternate or switch group. Each differentiating attribute is associated with a pointer to the field or information that distinguishes the track.

8.10.3.5 Attributes

The following attributes are descriptive:

Name	*Attribute*	*Description*
Temporal scalability	'tesc'	The track can be temporally scaled.
Fine-grain SNR scalability	'fgsc'	The track can be scaled in terms of quality.
Coarse-grain SNR scalability	'cgsc'	The track can be scaled in terms of quality.
Spatial scalability	'spsc'	The track can be spatially scaled.
Region-of-interest scalability	'resc'	The track can be region-of-interest scaled.
View scalability	'vwsc'	The track can be scaled in terms of number of views.

The following attributes are differentiating:

Name	*Attribute*	*Pointer*
Codec	'cdec'	Sample Entry (in SampleDescriptionBox of media track)
Screen size	'scsz'	Width and height fields of VisualSampleEntry.
Max packet size	'mpsz'	Maxpacketsize field in RtpHintSampleEntry
Media type	'mtyp'	Handlertype in HandlerBox (of media track)
Media language	'mela'	Language field in MediaHeaderBox
Bitrate	'bitr'	Total size of the samples in the track divided by the duration in the TrackHeaderBox
Frame rate	'frar'	Number of samples in the track divided by duration in the TrackHeaderBox
Number of views	'nvws'	Number of views in the track

Descriptive attributes characterize the tracks they modify, whereas differentiating attributes differentiate between tracks that belong to the same alternate or switch groups. The pointer of a differentiating attribute indicates the location of the information that differentiates the track from other tracks with the same attribute.

8.10.4 Track kind

8.10.4.1 Definition

Box Type: 'kind'
Container: UserDataBox of the corresponding TrackBox
Mandatory: No
Quantity: Zero or more

The KindBox labels a track with its role or kind.

It contains a URI, possibly followed by a value. If only a URI occurs, then the kind is defined by that URI; if a value follows, then the naming scheme for the value is identified by the URI.

More than one of these may occur in a track, with different contents but with appropriate semantics (e.g. two schemes that both define a kind that indicates sub-titles).

8.10.4.2 Syntax

```
aligned(8) class KindBox extends FullBox('kind', version = 0, 0) {
   utf8string schemeURI;
   utf8string value;
}
```

8.10.4.3 Semantics

`schemeURI` declares either the identifier of the kind, if no value follows, or the identifier of the naming scheme for the following value.

`value` is a name from the declared scheme

8.11 Metadata support

8.11.1 MetaBox

8.11.1.1 Definition

Box Type:	`'meta'`
Container:	File, Segment, `MovieBox`, `TrackBox`, `MovieFragmentBox` or `TrackFragmentBox`
Mandatory:	No
Quantity:	Zero or one (in File, `MovieBox`, and `TrackBox`), Zero or one (in Segment, `MovieFragmentBox` or `TrackFragmentBox`)

A common base structure is used to contain general untimed metadata. This structure is called the `MetaBox` as it was originally designed to carry metadata, i.e. data that is annotating other data. However, it is now used for a variety of purposes including the carriage of data that is not annotating other data, especially when present at 'file level'. The handling of metadata in movie fragments is described in 8.8.17.

The `MetaBox` is required to contain a `HandlerBox` indicating the structure or format of the `MetaBox` contents.

All other contained boxes are specific to the format specified by the `HandlerBox`.

The other boxes defined here may be defined as optional or mandatory for a given format. If they are used, then they shall take the form specified here. These optional boxes include a `DataInformationBox`, which documents other files in which metadata values (e.g. pictures) are placed, and an `ItemLocationBox`, which documents where in those files each item is located (e.g. in the common case of multiple pictures stored in the same file).

At most one `MetaBox` may occur at each of the file level, segment, movie level, or track level.

If an `ItemProtectionBox` occurs, then some or all of the metadata, including possibly the primary resource, may have been protected and be un-readable unless the protection system is taken into account.

NOTE The `MetaBox` is unusual in that it is a container box yet extends `FullBox`, not `Box`.

Metadata items are identified by `item_ID`. Within a given `MetaBox`, a given `item_ID` shall uniquely refer to a single item. When an item is updated in movie fragments, the `item_ID` refers to the latest received version.

Derived specifications may further restrict the criteria for uniqueness: unique among the `item_IDs` in both file and movie-level boxes, or unique within that set extended with the `track_ID` of the tracks in a movie box. The `item_ID` value of 0 should not be used, and shall not be used when the set is extended to include `track_IDs`.

There are three scopes for `item_IDs`: file and segments; `MovieBox` and `MovieFragmentBox`; and `TrackBox` and `TrackFragmentBox`. In other words, there shall be only one item with a given `item_ID` within a given scope (e.g. in the `TrackBox` and all `TrackFragmentBox` with the same `track_ID`).

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`.

8.11.2 XML boxes

8.11.2.1 Definition

Box Type: `'xml '` or `'bxml'`
Container: `MetaBox`
Mandatory: No
Quantity: Zero or one

When the primary data is in XML format as defined by the W3C Recommendation, *Extensible Markup Language (XML)* and it is desired that the XML be stored directly in the `MetaBox`, one of these forms may be used. The `BinaryXMLBox` may only be used when there is a single well-defined binarization of the XML for that defined format as identified by the handler.

The use of the `XMLBox` in new specifications is deprecated. The preferred technique is to use an item, identify it as a primary item, and identify its precise XML format in the `ItemInfoBox`.

Previous edition of this standard did not mandate the XML data to be null-terminated, while this edition does. Writers conformant to this edition shall use a null termination character. Readers should tolerate boxes missing this null termination character.

8.11.2.2 Syntax

```
aligned(8) class XMLBox
   extends FullBox('xml ', version = 0, 0) {
   utfstring xml;
}
aligned(8) class BinaryXMLBox
   extends FullBox('bxml', version = 0, 0) {
   unsigned int(8) data[];      // to end of box
}
```

8.11.2.3 Semantics

`xml` is a zero-terminated string containing the XML data.

`data` contains the encoded XML data.

8.11.3 Item location box

8.11.3.1 Definition

Box Type: `'iloc'`
Container: `MetaBox`

Mandatory: No
Quantity: Zero or one

The ItemLocationBox provides a directory of resources in this or other files, by locating their container, their offset within that container, and their length. Using byte offsets and lengths 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 place, re-adjusting offsets and 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 versions 1 and 2 of this box, the item_reference_index fields, respectively. These values shall be from the set {0, 4, 8}.

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

i) file_offset: by absolute byte offsets into the file or the payload of IdentifiedMediaDataBox referenced by data_reference_index; (construction_method == 0)

ii) idat_offset: by byte offsets into the ItemDataBox in the same MetaBox; neither the data_reference_index nor item_reference_index fields are used; (construction_method == 1)

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

The item_reference_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 in the order specified in this box. 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) 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 structure-data, 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 1 Extents can be interleaved with the chunks defined by the sample tables of tracks.

The offsets are relative to a data origin. That origin is determined as follows:

1) when the MetaBox is in a Movie Fragment, and the construction_method specifies a file offset, and the data reference indicates 'same file', the data origin is the first byte of the enclosing MovieFragmentBox (as for the default-base-is-moof flag in the TrackFragmentHeaderBox);

2) when the construction_method specifies a file offset and the data reference indicates DataEntryImdaBox or DataEntrySeqNumImdaBox, the data origin is the first byte of the payload of the corresponding IdentifiedMediaDataBox;

3) in all other cases when the construction_method specifies a file offset, the data origin is the beginning of the file identified by the data reference;

4) when the construction_method specifies offsets into the ItemDataBox, the data origin is the beginning of data[] in the ItemDataBox;

5) when the data reference specifies another item, the data origin is the first byte of the concatenated data (of all the extents) of that item;

NOTE 2 There are offset calculations in other parts of this file format based on the beginning of a box header; in contrast, item data offsets are calculated relative to the box payload.

The `data_reference_index` may take the value 0, indicating a reference into the same file as this structure-data, or an index into the data references in the `DataInformationBox` in the containing `MetaBox`, with value 1 indicating the first entry in the data reference list.

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 in the item itself 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 document, 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`, or version 2 when possible. Similarly, version 2 of this box should only be used when support for large `item_ID` values (exceeding 65535) is required or expected to be required.

NOTE 3 When `construction_method` 2 is used and one item needs to have an offset of 0 into another item, the `base_offset` field is set to 0.

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) || (version == 2)) {
      unsigned int(4)    index_size;
   } else {
      unsigned int(4)    reserved;
   }
   if (version < 2) {
      unsigned int(16)    item_count;
   } else if (version == 2) {
      unsigned int(32)    item_count;
   }
   for (i=0; i<item_count; i++) {
      if (version < 2) {
         unsigned int(16)    item_ID;
      } else if (version == 2) {
         unsigned int(32)    item_ID;
      }
      if ((version == 1) || (version == 2)) {
         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) || (version == 2)) && (index_size > 0)) {
            unsigned int(index_size*8)    item_reference_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 `item_reference_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 an index, with value 1 indicating the first entry, into the data references in the `DataInformationBox`.

`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 shall have the value 1 or greater

`item_reference_index` provides an index as defined for the construction method

`extent_offset` provides the absolute offset, in bytes from the data origin of the container, of this extent data. If `offset_size` is 0, `extent_offset` takes the value 0

`extent_length` provides the absolute length in bytes of this metadata item extent. If `length_size` is 0, `extent_length` takes the value 0. If the value is 0, then length of the extent is the length of the entire referenced container.

8.11.4 Primary item box

8.11.4.1 Definition

Box Type: `'pitm'`
Container: `MetaBox`
Mandatory: No
Quantity: Zero or one

For a given handler, the primary data may be one of the referenced items when it is desired that it be stored elsewhere, or divided into extents; or the primary metadata may be contained in the `MetaBox` (e.g. in an `XMLBox`). Either this box shall occur, or there shall be a box within the `MetaBox` (e.g. an `XMLBox`) containing the primary information in the format required by the identified handler.

8.11.4.2 Syntax

```
aligned(8) class PrimaryItemBox
      extends FullBox('pitm', version, 0) {
   if (version == 0) {
      unsigned int(16)   item_ID;
   } else {
      unsigned int(32)   item_ID;
   }
}
```

8.11.4.3 Semantics

item_ID is the identifier of the primary item, which shall be the identifier of an item in the MetaBox containing the PrimaryItemBox. Version 1 should only be used when large item_ID values (exceeding 65535) are required or expected to be required.

8.11.5 Item protection box

8.11.5.1 Definition

Box Type: 'ipro'
Container: MetaBox
Mandatory: No
Quantity: Zero or one

The ItemProtectionBox provides an array of item protection information, for use by the ItemInfoBox.

8.11.5.2 Syntax

```
aligned(8) class ItemProtectionBox
      extends FullBox('ipro', version = 0, 0) {
   unsigned int(16) protection_count;
   for (i=1; i<=protection_count; i++) {
      ProtectionSchemeInfoBox   protection_information;
   }
}
```

8.11.6 Item information box

8.11.6.1 Definition

Box Type: 'iinf'
Container: MetaBox
Mandatory: No
Quantity: Zero or one

The ItemInfoBox provides extra information about selected items, including symbolic ('file') names. It may optionally occur, but if it does, it shall 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. The item_name shall be a valid URL (e.g. a simple name, or path name) and shall not be an absolute URL.

Four 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 FilePartitionBox, which is defined for source file partitionings and applies to file delivery transmissions. Versions 2 and 3 provide an alternative structure in which metadata item types are indicated by a 32-bit registered or defined code (typically a four character code); two of these codes are defined to indicate a MIME type or metadata typed by a URI. Version 2 supports 16-bit item_ID values, whereas version 3 supports 32-bit item_ID values.

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, shall be supplied for the content_encoding, before the indication of extension_type.

If file delivery item information is needed and a version 2 or 3 ItemInfoEntry is used, then the file delivery information is stored as a separate item of type 'fdel' that is also linked by an item reference from the item, to the file delivery information, of type `'fdel'`. There shall be exactly one such reference if file delivery information is needed.

It is possible that there are valid URI forms for MPEG-7 metadata as defined in ISO/IEC 15938-1 (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.

Version 1 of `ItemInfoBox` should only be used when support for a large number of `itemInfoEntries` (exceeding 65535) is required or expected to be required.

The `flags` field of `ItemInfoEntry` with `version` greater than or equal to 2 is specified as follows:

— (`flags` & 1) equal to 1 indicates that the item is not intended to be a part of the presentation. .

— (`flags` & 1) equal to 0 indicates that the item is intended to be a part of the presentation.

8.11.6.2 Syntax

```
aligned(8) class ItemInfoExtension(unsigned int(32) extension_type)
{
}
aligned(8) class FDItemInfoExtension() extends ItemInfoExtension ('fdel') {
   utf8string content_location;
   utf8string 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, flags) {
   if ((version == 0) || (version == 1)) {
      unsigned int(16) item_ID;
      unsigned int(16) item_protection_index;
      utf8string item_name;
      utf8string content_type;
      utf8string content_encoding; //optional
   }
   if (version == 1) {
      unsigned int(32) extension_type; //optional
      ItemInfoExtension(extension_type); //optional
   }
   if (version >= 2) {
      if (version == 2) {
         unsigned int(16) item_ID;
      } else if (version == 3) {
         unsigned int(32) item_ID;
      }
      unsigned int(16) item_protection_index;
      unsigned int(32) item_type;
      utf8string item_name;
      if (item_type=='mime') {
         utf8string content_type;
         utf8string content_encoding; //optional
      } else if (item_type == 'uri ') {
         utf8string item_uri_type;
      }
   }
}
aligned(8) class ItemInfoBox
      extends FullBox('iinf', version, 0) {
   if (version == 0) {
      unsigned int(16)   entry_count;
   } else {
      unsigned int(32) 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 `XMLBox`) 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 index, with value 1 indicating the first entry, into the `ItemProtectionBox` defining the protection applied to this item (the first box in the `ItemProtectionBox` has the index 1).

`item_name` is the 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 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 an absolute URI, that is used as a type indicator.

`content_encoding` optionally indicates 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 four character code that identifies the extension fields of version 1 with respect to version 0 of the Item information entry.

`content_location` contains the URI of the file as defined in HTTP/1.1 (IETF RFC 2616[8]).

`content_MD5` contains an MD5 digest of the file. See HTTP/1.1 (IETF RFC 2616[8]) and IETF RFC 1864[7].

`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. 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[2] for more details on file groups.

8.11.7 Additional metadata container box

This box has been deprecated and is no longer defined in this document.

8.11.8 Metabox Relation box

This box has been deprecated and is no longer defined in this document.

8.11.9 URL forms for MetaBoxes

When a `MetaBox` is used, then URLs may be used to refer to items in the `MetaBox`, either using an absolute URL, or using a relative URL.

When interpreting data that is in the context of a `MetaBox`, the items in the `MetaBox` are treated as *shadowing* files in the same location as that from which file containing the `MetaBox` (the 'container file') came, i.e. the contents of the item is the same as the file located at the URL obtained by resolving an absolute URL from the absolute container file URL used as base URL and from the `item_name` (if not empty) used as relative URL. This shadowing means that a reference to another *file* in the same location as the container file may be resolved to an *item* within the container file itself.

Items can be addressed within the container file by appending a fragment to the URL for the container file itself as specified in Annex C.

Consider the following example:

```
<http://a.com/d/v.qrv#item_name=tree.html*branch1>.
```

We assume that `v.qrv` is a file with a `MetaBox` at the file level. First, the client strips the fragment and fetches `v.qrv` from a.com using HTTP. It then inspects the top-level `MetaBox` and adds the items in it, logically, to its cache of the directory “d” on a.com. It then re-forms the URL as `<http://a.com/d/tree.html#branch1>`. See that the fragment has been elevated to a full file name, and the first “*” has been transformed back into a “#”. The client then either finds an item named `tree.html` in the `MetaBox`, or fetches `tree.html` from `a.com`, and it then finds the anchor “`branch1`” within `tree.html`. If within that html, a file was referenced using a relative URL, e.g. “`flower.gif`”, then the client converts this to an absolute URL using the normal rules: `<http://a.com/d/flower.gif>` and again it checks to see if `flower.gif` is a named item (and hence shadowing a separate file of this name), and then if it is not, fetches `flower.gif` from `a.com`.

8.11.10 Static metadata

8.11.10.1 General

This subclause defines the storage of static (un-timed) metadata in the ISO file format family.

Reader support for metadata in general is optional, and therefore it is also optional for the formats defined here or elsewhere, unless made mandatory by a derived specification.

8.11.10.2 Simple textual

There is existing support for simple textual tags in the form of the user-data boxes; currently only one is defined – the copyright notice. Other metadata is permitted using this simple form if:

a) it uses a registered box-type or it uses the UUID escape (the latter is permitted today);

b) it uses a registered tag, the equivalent MPEG-7 construct must be documented as part of the registration.

8.11.10.3 Other forms

When other forms of metadata are desired, then a `MetaBox` as defined above may be included at the appropriate level of the document. If the document is intended to be primarily a metadata document per se, then the `MetaBox` is at file level. If the metadata annotates an entire presentation, then the `MetaBox` is at the movie level; an entire stream, at the track level.

8.11.10.4 MPEG-7 metadata

When MPEG-7 metadata as defined in ISO/IEC 15938-1 is stored in `MetaBox`es the requirements of this clause apply.

1) When the handler-type is `'mp7t'` the metadata shall be in textual form, shall conform to ISO/IEC 15938-1, and shall use the encoding defined in ISO/IEC 10646 (technically identical to the Unicode standard[28]).

2) When the handler-type is `'mp7b'` the metadata shall be in binary form compressed in the BIM format as specified in ISO/IEC 23001-1. In this case, the `BinaryXMLBox` contains the configuration information immediately followed by the binarized XML.

3) When the format is textual, there shall be either another box in the `MetaBox`, called `'xml '`, which contains the textual MPEG-7 document, or there shall be a `PrimaryItemBox` identifying the item containing the MPEG-7 XML.

4) When the format is binary, there shall be either another box in the `MetaBox`, called `'bxml'`, which contains the binary MPEG-7 document, or a `PrimaryItemBox` identifying the item containing the MPEG-7 binarized XML.

5) If an MPEG-7 box is used at the file level, then the brand `'mp71'` should be a member of the compatible-brands list in the `FileTypeBox`.

8.11.11 Item data box

8.11.11.1 Definition

Box Type: `'idat'`
Container: `MetaBox`
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 metadata

8.11.12 Item reference box

8.11.12.1 Definition

Box Type: `'iref'`
Container: `MetaBox`
Mandatory: No
Quantity: Zero or one

The `ItemReferenceBox` 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 `SingleItemTypeReferenceBox`, 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; within a given array, a given value shall occur at most once. Other structures in the file formats index through these arrays; index values start at 1.

All these single item type reference boxes are then collected into the ItemReferenceBox. 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. Version 1 of ItemReferenceBox with SingleItemReferenceBoxLarge should only be used when large from_item_ID or to_item_ID values (exceeding 65535) are required or expected to be required.

NOTE 1 This design makes it fairly easy to find all the references of a specific type, or from a specific item.

An item reference of type 'font' may be used to indicate that an item uses fonts carried/defined in the referenced item.

NOTE 2 Other structures index through the array of item references and hence position and order of them can be significant.

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 SingleItemTypeReferenceBoxLarge(referenceType) extends Box(referenceType)
{
   unsigned int(32) from_item_ID;
   unsigned int(16) reference_count;
   for (j=0; j<reference_count; j++) {
      unsigned int(32) to_item_ID;
   }
}
aligned(8) class ItemReferenceBox extends FullBox('iref', version, 0) {
   if (version==0) {
      SingleItemTypeReferenceBox              references[];
   } else if (version==1) {
      SingleItemTypeReferenceBoxLarge    references[];
   }
}
```

8.11.12.3 Semantics

reference_type contains an indication of the type of the reference

from_item_ID contains the item_ID of the item that refers to other items

reference_count is the number of references

to_item_ID contains the item_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 metadata item of type 'auvd' (auxiliary video descriptor); the data of that item shall be 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 metadata items (e.g. one for parallax info and one for depth, in the case that the same stream serves).

8.11.14 Item properties box

8.11.14.1 Definition

Box Type: 'iprp'
Container: MetaBox ('meta')

Mandatory: No
Quantity: Zero or one

The ItemPropertiesBox enables the association of any item with an ordered set of item properties. Item properties are small data records.

The ItemPropertiesBox consists of two parts: ItemPropertyContainerBox that contains an implicitly indexed list of item properties, and one or more ItemPropertyAssociationBox(es) that associate items with item properties.

Each item property is a Box or FullBox. The boxtype of the item property specifies the property type. The FreeSpaceBox may occur in the ItemPropertyContainerBox; it has no meaning, and should not be associated with any item.

NOTE Item Properties are based on the box format documented in 4.2 which means that boxes with an extended_type and the type field set to 'uuid' are permissible as item properties.

Each property association may be marked as either essential or non-essential. A reader shall not process an item that is associated with a property that is not recognized or not supported by the reader and that is marked as essential to the item. A reader may ignore an associated item property that is marked non-essential to the item.

Specifications deriving from this document may specify property types and the respective item property box definitions as well as constraints and requirements for the property associations.

When defining item properties, it is recommended that they be small. When large data records need to be associated with an item, a separate item and item reference are more suitable.

Each ItemPropertyAssociationBox shall be ordered by increasing item_ID, and there shall be at most one occurrence of a given item_ID, in the set of ItemPropertyAssociationBox boxes. The version 0 should be used unless 32-bit item_ID values are needed; similarly, flags should be equal to 0 unless there are more than 127 properties in the ItemPropertyContainerBox. There shall be at most one ItemPropertyAssociationBox with a given pair of values of version and flags.

8.11.14.2 Syntax

```
aligned(8) class ItemProperty(property_type)
   extends Box(property_type)
{
}

aligned(8) class ItemFullProperty(property_type, version, flags)
   extends FullBox(property_type, version, flags)
{
}

aligned(8) class ItemPropertyContainerBox
   extends Box('ipco')
{
   Box properties[];    // boxes derived from
      // ItemProperty or ItemFullProperty, or FreeSpaceBox(es)
      // to fill the box
}

aligned(8) class ItemPropertyAssociationBox
   extends FullBox('ipma', version, flags)
{
   unsigned int(32) entry_count;
   for(i = 0; i < entry_count; i++) {
      if (version < 1)
         unsigned int(16)   item_ID;
      else
         unsigned int(32)   item_ID;
      unsigned int(8) association_count;
      for (i=0; i<association_count; i++) {
         bit(1) essential;
         if (flags & 1)
            unsigned int(15) property_index;
         else
            unsigned int(7) property_index;
      }
   }
}

aligned(8) class ItemPropertiesBox
      extends Box('iprp') {
   ItemPropertyContainerBox property_container;
   ItemPropertyAssociationBox association[];
 }
```

8.11.14.3 Semantics

`item_ID` identifies the item with which properties are associated

`essential` when set to 1 indicates that the associated property is essential to the item, otherwise it is non-essential

`property_index` is either 0 indicating that no property is associated (the essential indicator shall also be 0), or is the 1-based index (counting all boxes, including `FreeSpace` boxes) of the associated property box in the `ItemPropertyContainerBox` contained in the same `ItemPropertiesBox`.

8.11.15 Brand item property

8.11.15.1 Definition

Box Type:	`'brnd'`
Property Type:	Descriptive item property
Container:	`ItemPropertyContainerBox`
Mandatory (per an item):	No
Quantity (per an item):	Zero or one

The payload of `BrandProperty` has the syntax of the `GeneralTypeBox`.

The content of an instance of `BrandProperty` shall be such that it would apply as the content of `FileTypeBox`, if the following modifications were made to the file:

— All tracks of the file are removed.

— All items except the following are removed:

  — The item associated with this `BrandProperty`, referred to as currItem.

  — The items, referred to as referencedItemsList, that currItem directly or indirectly references to, as indicated in the `ItemReferenceBox`.

— When the current primary item is not a member of the set including currItem and referencedItems, currItem is set to be the primary item in the `PrimaryItemBox`.

8.11.15.2 Syntax

```
aligned(8) class BrandProperty extends GeneralTypeBox ('brnd')
{ }
```

8.12 Support for protected streams

8.12.1 Overview

This subclause documents the file-format transformations which are used for protected content. These transformations can be used under several circumstances:

— They shall be used when the content has been transformed (e.g. by encryption) in such a way that it can no longer be decoded by the normal decoder;

— They may be used when the content should only be decoded when the protection system is understood and implemented.

The transformation functions by *encapsulating* the original media declarations. The encapsulation changes the four character-code of the sample entries, so that protection-unaware readers see the media stream as a new stream format.

Because the format of a sample entry varies with media-type, a different encapsulating four character-code is used for each media type (audio, video, text etc.). They are shown in Table 5.

Table 5 — Protected sample-entry codes

Stream (Track) Type	Sample-Entry Code	SampleEntry Class
Video	`encv`	`VisualSampleEntry`
Audio	`enca`	`AudioSampleEntry or AudioSampleEntryV1`
Metadata	`encm`	`MetaDataSampleEntry`
Text	`enct`	`SimpleTextSampleEntry`
Subtitle	`encu`	`XMLSubtitleSampleEntry`
System[a]	`encs`	
Font	`encf`	`FontSampleEntry`
Haptics	`encp`	`HapticSampleEntry`
Volumetric visual	`enc3`	`VolumetricVisualSampleEntry`

[a] System streams are defined in ISO/IEC 14496-14[22].

The transformed sample entry type shall only be used with the indicated sample entry classes, or classes derived from them that add only boxes (but not fields).

A protected sample entry is defined as using one of the preceding sample entry codes, and the following transformation procedure:

1) The four character code of the sample description is replaced with a four character code indicating protection encapsulation: these codes vary only by media-type. For example, `'mp4v'` is replaced with `'encv'` and `'mp4a'` is replaced with `'enca'`.

2) A `ProtectionSchemeInfoBox` (*defined below*) is added to the sample description, leaving all other boxes unmodified.

3) The original sample entry type (four character code) is stored within the `ProtectionSchemeInfoBox`, in a new box called the `OriginalFormatBox` (defined below);

There are then three methods for signalling the nature of the protection, which may be used individually or in combination.

1) When MPEG-4 systems is used, then IPMP shall be used to signal that the streams are protected.

2) IPMP descriptors may also be used outside the MPEG-4 systems context using boxes containing IPMP descriptors.

3) The protection applied may also be described using the scheme type and information boxes.

When IPMP is used outside of MPEG-4 systems, then a ‘global’ `IPMPControlBox` may also occur within the `MovieBox`.

NOTE When MPEG-4 systems is used, an MPEG-4 systems terminal can effectively treat, for example, `'encv'` with an original format of `'mp4v'` exactly the same as `'mp4v'`, by using the IPMP descriptors.

8.12.2 Protection scheme information box

8.12.2.1 Definition

Box Types: `'sinf'`
Container: Protected Sample Entry, or `ItemProtectionBox`
Mandatory: Yes
Quantity: One or More

The `ProtectionSchemeInfoBox` contains all the information required both to understand the encryption transform applied and its parameters, and also to find other information such as the kind and location of the key management system. It also documents the original (unencrypted) format of the media. The `ProtectionSchemeInfoBox` is a container Box. It is mandatory in a sample entry that uses a code indicating a protected stream.

When used in a protected sample entry, this box shall contain the `OriginalFormatBox` to document the original format. At least one of the following signalling methods shall be used to identify the protection applied:

a) MPEG-4 systems with IPMP: no other boxes, when IPMP descriptors in MPEG-4 systems streams are used;

b) Scheme signalling: a `SchemeTypeBox` and `SchemeInformationBox`, when these are used (either both shall occur, or neither).

At least one `ProtectionSchemeInfoBox` shall occur in a protected sample entry. When more than one occurs, they are equivalent, alternative, descriptions of the same protection. Readers should choose one to process.

8.12.2.2 Syntax

```
aligned(8) class ProtectionSchemeInfoBox(fmt) extends Box('sinf') {
   OriginalFormatBox(fmt)    original_format;

   SchemeTypeBox          scheme_type_box;      // optional
   SchemeInformationBox    info;                  // optional
}
```

8.12.3 Original format box

8.12.3.1 Definition

Box Types: `'frma'`
Container: `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or
    `CompleteTrackInfoBox`
Mandatory: Yes when used in a protected sample entry, in a restricted sample entry, or
    in a sample entry for an incomplete track.
Quantity: Exactly one.

The `OriginalFormatBox` contains the four character code of the original un-transformed sample description.

8.12.3.2 Syntax

```
aligned(8) class OriginalFormatBox(codingname) extends Box ('frma') {
   unsigned int(32)   data_format = codingname;
         // format of decrypted, encoded data (in case of protection)
         // or un-transformed sample entry (in case of restriction
         // and complete track information)
}
```

8.12.3.3 Semantics

`data_format` is the four character code of the original un-transformed sample entry (e.g. `'mp4v'` if the stream contains protected or restricted MPEG-4 visual material).

8.12.4 IPMPInfoBox

This box has been deprecated and is no longer defined in this document.

8.12.5 IPMP control box

This box has been deprecated and is no longer defined in this document.

8.12.6 Scheme type box

8.12.6.1 Definition

Box Types: `'schm'`
Container: `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or `SRTPProcessBox`
Mandatory: No
Quantity: Zero or one in `'sinf'`, depending on the protection structure; Exactly one in `'rinf'` and `'srpp'`

The `SchemeTypeBox` identifies the protection or restriction scheme.

8.12.6.2 Syntax

```
aligned(8) class SchemeTypeBox extends FullBox('schm', 0, flags) {
   unsigned int(32) scheme_type; // 4CC identifying the scheme
   unsigned int(32) scheme_version; // scheme version
   if (flags & 0x000001) {
      utf8string scheme_uri; // browser uri
   }
}
```

8.12.6.3 Semantics

`scheme_type` is the code defining the protection or restriction scheme, normally expressed as a four character code;

`scheme_version` is the version of the scheme (used to create the content)

`scheme_URI` is an absolute URI allowing for the option of directing the user to a web-page if they do not have the scheme installed on their system.

8.12.7 Scheme information box

8.12.7.1 Definition

Box Types: `'schi'`
Container: `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or `SRTPProcessBox`

Mandatory: No
Quantity: Zero or one

The SchemeInformationBox is a container Box that is only interpreted by the scheme being used. Any information the encryption or restriction system needs is stored here. The content of this box is a series of boxes whose type and format are defined by the scheme declared in the SchemeTypeBox.

8.12.7.2 Syntax

```
aligned(8) class SchemeInformationBox extends Box('schi') {
   Box   scheme_specific_data[];
}
```

8.12.8 Scramble Scheme Information Box

8.12.8.1 Definition

Box Types:	'scrb'

Container: Sample entry or ItemPropertyContainerBox
Mandatory: Yes, if the media sample(s) associated with the sample entry or the associated item(s) use a scrambling scheme
Quantity: One or More

In the context of this document, a scrambling operation on media stream is the process of modifying binary patterns in a compressed media stream without modifying the media bitstream syntax (i.e. decoding without reverting the scramble is syntactically correct). The ScrambleSchemeInfoBox contains all the information required both to understand the scrambling operation applied and its parameters, and also to find other information such as the kind and location of the key management system. The ScrambleSchemeInfoBox is a container box.

It is mandatory in a sample entry for which media samples use a scrambling scheme resulting in media content compliant with the requirements of the media format identified by the sample entry type of the track, if the track is not a transformed media track, or by the untransformed sample entry type, otherwise.

For an item consisting of media using a scrambling scheme resulting in media content compliant with the requirements of the media format identified by the item type, this property shall be present in the ItemPropertyContainerBox and the protected item shall have this property listed as a non-essential property.

This box shall not be used for any scrambling scheme resulting in a cyphered media stream not compatible with the requirements of the media format identified by the sample entry type or item type.

NOTE The ScrambleSchemeInfoBox is identical to the ProtectionSchemeInfoBox except that (a) it uses the four-character code 'scrb' and (b) does not contain an OriginalFormatBox.

8.12.8.2 Syntax

```
aligned(8) class ScrambleSchemeInfoBox extends Box('scrb') {
   SchemeTypeBox scheme_type_box;
   SchemeInformationBox info; // optional
}
```

8.13 File delivery format support

8.13.1 Overview

Files intended for transmission over ALC/LCT or FLUTE are stored as items in a top-level MetaBox. The ItemLocationBox specifies the actual storage location of each item within the container file as well as the file size of each item. File name, content type (MIME type), etc., of each item are provided by version 1 of the ItemInfoBox.

Pre-computed FEC reservoirs are stored as additional items in the MetaBox. 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 PartitionEntry located in the FDItemInformationBox.

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 PartitionEntry located in the FDItemInformationBox.

See subclause 9.2 for more details on the usage of the file delivery format.

8.13.2 FD item information box

8.13.2.1 Definition

Box Type: 'fiin'
Container: MetaBox
Mandatory: No
Quantity: Zero or one

The FDItemInformationBox 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. It is possible to provide multiple entries for one source file (identified by its item_ID) if alternative FEC encoding schemes or partitionings are used in the file. All partition entries are implicitly numbered and the first entry has number 1.

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
}
```

8.13.2.3 Semantics

entry_count provides a count of the number of entries in the following array.

The semantics of the boxes are described where the boxes are documented.

8.13.3 File partition box

8.13.3.1 Definition

Box Type: 'fpar'
Container: PartitionEntry
Mandatory: Yes
Quantity: Exactly one

The FilePartitionBox identifies the source file and provides a partitioning of that file into source blocks and symbols. Further information about the source file, e.g., filename, content location and group IDs, is contained in the ItemInfoBox, where the ItemInfoEntry corresponding to the item_ID of the source file is of version 1 and includes a `FDItemInfoExtension`. Version 1 of `FilePartitionBox` should only be used when support for large `item_ID` or `entry_count` values (exceeding 65535) is required or expected to be required.

8.13.3.2 Syntax

```
aligned(8) class FilePartitionBox
      extends FullBox('fpar', version, 0) {
   if (version == 0) {
      unsigned int(16)   item_ID;
   } else {
      unsigned int(32)   item_ID;
   }
   unsigned int(16)   packet_payload_size;
   unsigned int(8)    reserved = 0;
   unsigned int(8)    FEC_encoding_ID;
   unsigned int(16)   FEC_instance_ID;
   unsigned int(16)   max_source_block_length;
   unsigned int(16)   encoding_symbol_length;
   unsigned int(16)   max_number_of_encoding_symbols;
   base64string       scheme_specific_info;
   if (version == 0) {
      unsigned int(16)   entry_count;
   } else {
      unsigned int(32)   entry_count;
   }
   for (i=1; i <= entry_count; i++) {
      unsigned int(16)   block_count;
      unsigned int(32)   block_size;
   }
}
```

8.13.3.3 Semantics

`item_ID` references the item in the `ItemLocationBox` that the file partitioning applies to.

`packet_payload_size` gives the target ALC/LCT or FLUTE packet payload size of the partitioning algorithm. Note that UDP packet payloads are larger, as they also contain ALC/LCT or FLUTE headers.

`FEC_encoding_ID` shall identify the FEC encoding scheme using a "Reliable Multicast Transport (RMT) FEC Encoding ID" declared at IANA, as defined in IETF RFC 5052. Note that i) value zero corresponds to the "Compact No-Code FEC scheme" also known as "Null-FEC" (IETF RFC 3695[6]); ii) value one corresponds to the “MBMS FEC” (3GPP TS 26.346[2]); 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` shall provide 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 IETF RFC 5052 for further details.

`max_source_block_length` gives the maximum number of source symbols per source block.

`encoding_symbol_length` gives the size (in bytes) of one encoding symbol. All encoding symbols of one item have the same length, except the last symbol which may be shorter.

`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 IETF RFC 5052. For those FEC schemes in which the maximum number of encoding symbols is not relevant, the semantics of this field is unspecified.

`scheme_specific_info` is the scheme-specific object transfer information (FEC-OTI-Scheme-Specific-Info). The definition of the information depends on the FEC encoding ID.

`entry_count` gives the number of entries in the list of (`block_count`, `block_size`) pairs that provides a partitioning of the source file. Starting from the beginning of the file, each entry indicates how the next segment of the file is divided into source blocks and source symbols.

`block_count` indicates the number of consecutive source blocks of size `block_size`.

`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[2]) the padding may extend across multiple symbols but the size of padding should never be more than encoding_symbol_length.

8.13.4 FEC reservoir box

8.13.4.1 Definition

Box Type: `'fecr'`
Container: `PartitionEntry`
Mandatory: No
Quantity: Zero or One

The `FECReservoirBox` associates the source file identified in the `FilePartitionBox` with FEC reservoirs stored as additional items. It contains a list that starts with the first FEC reservoir associated with the first source block of the source file and continues sequentially through the source blocks of the source file. Version 1 of `FECReservoirBox` should only be used when support for large `item_ID` values and `entry_count` (exceeding 65535) is required or expected to be required.

8.13.4.2 Syntax

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

8.13.4.3 Semantics

`entry_count` gives the number of entries in the following list. An entry count here should match the total number of blocks in the corresponding `FilePartitionBox`.

`item_ID` indicates the location of the FEC reservoir associated with a source block.

`symbol_count` indicates the number of repair symbols contained in the FEC reservoir.

8.13.5 FD session group box

8.13.5.1 Definition

Box Type: `'segr'`
Container: `FDItemInformationBox`

Mandatory: No
Quantity: Zero or One

The FDSessionGroupBox is optional, although it is mandatory for files containing more than one FD hint track. It contains a list of sessions as well as all file groups and hint tracks that belong to each session. An FD session sends simultaneously over all FD hint tracks (channels) that are listed in the FD session group box for a particular FD session.

Only one session group should be processed at any time. The first listed hint track in a session group specifies the base channel. If the server has no preference between the session groups, the default choice should be the first session group. The group IDs of all file groups containing the files referenced by the hint tracks shall be included in the list of file groups. The file group IDs can in turn be translated into file group names (using the group ID to name box) that can be included by the server in FDTs.

8.13.5.2 Syntax

```
aligned(8) class FDSessionGroupBox extends Box('segr') {
   unsigned int(16)   num_session_groups;
   for(i=0; i < num_session_groups; i++) {
      unsigned int(8)   entry_count;
      for (j=0; j < entry_count; j++) {
         unsigned int(32)   group_ID;
      }
      unsigned int(16) num_channels_in_session_group;
      for(k=0; k < num_channels_in_session_group; k++) {
         unsigned int(32) hint_track_ID;
      }
   }
}
```

8.13.5.3 Semantics

num_session_groups specifies the number of session groups.

entry_count gives the number of entries in the following list comprising all file groups that the session group complies with. The session group contains all files included in the listed file groups as specified by the item information entry of each source file. The FDT for the session group should only contain those groups that are listed in this structure.

group_ID indicates a file group that the session group complies with.

num_channels_in_session_groups specifies the number of channels in the session group. The value of num_channels_in_session_groups shall be a positive integer.

hint_track_ID specifies the track identifier of the FD hint track belonging to a particular session group. Note that one FD hint track corresponds to one LCT channel.

8.13.6 Group ID to name box

8.13.6.1 Definition

Box Type: 'gitn'
Container: FDItemInformationBox
Mandatory: No
Quantity: Zero or One

The GroupIdToNameBox associates file group names to file group IDs used in the version 1 item information entries in the ItemInfoBox.

8.13.6.2 Syntax

```
aligned(8) class GroupIdToNameBox
      extends FullBox('gitn', version = 0, 0) {
```

```
   unsigned int(16)   entry_count;
   for (i=1; i <= entry_count; i++) {
      unsigned int(32)   group_ID;
      utf8string         group_name;
   }
}
```

8.13.6.3 Semantics

`entry_count` gives the number of entries in the following list.

`group_ID` indicates a file group.

`group_name` is the file group name.

8.13.7 File reservoir box

8.13.7.1 Definition

Box Type: `'fire'`
Container: `PartitionEntry`
Mandatory: No
Quantity: Zero or One

The `FileReservoirBox` associates the source file identified in the `FilePartitionBox` 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. Version 1 of `FileReservoirBox` should only be used when support for large `item_ID` or `entry_count` values (exceeding 65535) is required or expected to be required.

8.13.7.2 Syntax

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

`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.

8.14 Sub tracks

8.14.1 Overview

Sub tracks are used to assign parts of tracks to alternate and switch groups in the same way as (entire) tracks can be assigned to alternate and switch groups to indicate whether those tracks are alternatives to each other and whether it makes sense to switch between them during a session. Sub tracks are suitable for layered media, e.g., SVC and MVC, where media alternatives often are incommensurate with track structures. By defining alternate and switch groups at sub-track level it is possible to use existing rules for media selection and switching for such layered codecs. The over-all syntax is generic for all kinds of media and backward compatible with track-level definitions. Sub-track level alternate and switch groups use the same numbering as track level groups. The numberings are global over all tracks such that groups can be defined across track and sub-track boundaries.

In order to define sub tracks, media-specific definitions are required. Definitions for SVC and MVC are specified in the AVC file format (ISO/IEC 14496-15[16]). Another way is to define sample groups and map them to sub tracks using the `SubTrackSampleGroupBox` defined here. The syntax can also be extended to include other media-specific definitions.

For each sub track that shall be defined a SubTrackBox shall be included in the UserDataBox of the corresponding track. The SubTrackBox contains objects that define and provide information about a sub track in the same track. The `TrackSelectionBox` for this same track is already located here.

8.14.2 Backward compatibility

The default is to assign alternate and switch groups to 0 (zero) for (entire) tracks, which means that there is no information on alternate and/or switch groups for those (entire) tracks. However, file readers that are aware of sub-track definitions will be able to find sub-track information on alternate and switch groups even if the track indication is set to 0. This way it is possible to indicate that a file can be used by legacy readers by including the appropriate brand in the `FileTypeBox`. A file creator that requires a reader to be aware of sub-track information should not include legacy brands.

The same method of assigning sub track information can also be applied if all parts of a track except a sub track belong to the same alternate or switch group. Then the overall definitions can be made on track level as usual and specific assignments can be made at sub-track level. For sub tracks without specific assignments, track level assignments apply by default. As before, if a file creator requires a reader to be aware of sub-track information it should not include legacy brands (which would otherwise indicate that sub track information can be skipped).

8.14.3 Sub track box

8.14.3.1 Definition

Box Type: `'strk'`
Container: `UserDataBox` of the corresponding `TrackBox`
Mandatory: No
Quantity: Zero or more

This box contains objects that define and provide information about a sub track in the present track.

8.14.3.2 Syntax

```
aligned(8) class SubTrackBox extends Box('strk') {
}
```

8.14.4 Sub track information box

8.14.4.1 Definition

Box Type: `'stri'`
Container: `SubTrackBox`
Mandatory: Yes
Quantity: One

8.14.4.2 Syntax

```
aligned(8) class SubTrackInformationBox
   extends FullBox('stri', version = 0, 0){
   template int(16)   switch_group = 0;
   template int(16)   alternate_group = 0;
   template unsigned int(32)   sub_track_ID = 0;
   unsigned int(32)   attribute_list[];   // to the end of the box
}
```

8.14.4.3 Semantics

`switch_group` is an integer that specifies a group or collection of tracks and/or sub tracks. If this field is 0 (default value), then there is no information on whether the sub track can be used for switching during playing or streaming. If this integer is not 0 it shall be the same for tracks and/or sub tracks that can be used for switching between each other. Tracks that belong to the same switch group shall belong to the same alternate group. A switch group may have only one member.

`alternate_group` is an integer that specifies a group or collection of tracks and/or sub tracks. If this field is 0 (default value), then there is no information on possible relations to other tracks/sub-tracks. If this field is not 0, it should be the same for tracks/sub-tracks that contain alternate data for one another and different for tracks/sub-tracks belonging to different such groups. Only one track/sub-track within an alternate group should be played or streamed at any one time.

`sub_track_ID` is an integer. A non-zero value uniquely identifies the sub track locally within the track. A zero value (default) means that `sub_track_ID` is not assigned.

`attribute_list` is a list, to the end of the box, of attributes. The attributes in this list should be used as descriptions of sub tracks or differentiating criteria for tracks and sub tracks in the same alternate or switch group.

The following attributes are descriptive:

Name	*Attribute*	*Description*
Temporal scalability	`'tesc'`	The sub-track can be temporally scaled.
Fine-grain SNR scalability	`'fgsc'`	The sub-track can be scaled in terms of quality.
Coarse-grain SNR scalability	`'cgsc'`	The sub-track can be scaled in terms of quality.
Spatial scalability	`'spsc'`	The sub-track can be spatially scaled.
Region-of-interest scalability	`'resc'`	The sub-track can be region-of-interest scaled.
View scalability	`'vwsc'`	The sub-track can be scaled in terms of number of views.

The following attributes are differentiating:

Name	*Attribute*	*Pointer*
Bitrate	`'bitr'`	Total size of the samples in the sub track divided by the duration in the `TrackHeaderBox`
Frame rate	`'frar'`	Number of samples in the sub track divided by duration in the `TrackHeaderBox`
Number of views	`'nvws'`	Number of views in the sub track

8.14.5 Sub track definition box

8.14.5.1 Definition

Box Type: `'strd'`
Container: `SubTrackBox`

Mandatory: Yes
Quantity: One

This box contains objects that provide a definition of the sub track.

8.14.5.2 Syntax

```
aligned(8) class SubTrackDefinitionBox extends Box('strd') {
}
```

8.14.6 Sub track sample group box

8.14.6.1 Definition

Box Type: `'stsg'`
Container: `SubTrackDefinitionBox`
Mandatory: No
Quantity: Zero or more

This box defines a sub track as one or more sample groups by referring to the corresponding sample group descriptions describing the samples of each group.

8.14.6.2 Syntax

```
aligned(8) class SubTrackSampleGroupBox
   extends FullBox('stsg', 0, 0){
   unsigned int(32) grouping_type;
   unsigned int(16) item_count;
   for(i = 0; i< item_count; i++)
      unsigned int(32)   group_description_index;
}
```

8.14.6.3 Semantics

`grouping_type` is an integer that identifies the sample grouping. The value shall be the same as in the corresponding `SampleToGroupBox` and `SampleGroupDescriptionBox`.

`item_count` counts the number of sample groups listed in this box.

`group_description_index` is an integer that gives the index of the sample group entry which describes the samples in the group.

8.15 Post-decoder requirements on media

8.15.1 General

In order to handle situations where the file author requires certain actions on the player or renderer, this subclause specifies a mechanism that enables players to simply inspect a file to find out such requirements for rendering a bitstream and stops legacy players from decoding and rendering files that require further processing. The mechanism applies to any type of video codec. In particular it applies to AVC and for this case specific signalling is defined in the AVC file format (ISO/IEC 14496-15[16]) that allows a file author to list occurring SEI message IDs and distinguish between required and non-required actions for the rendering process.

The mechanism is similar to the content protection transformation where sample entries are hidden behind generic sample entries, `'encv'`, `'enca'`, etc., indicating encrypted or encapsulated media. The analogous mechanism for restricted video uses a transformation with the generic sample entry `'resv'`. The method may be applied when the content should only be decoded by players that present it correctly.

8.15.2 Restricted sample entry transformation

A restricted sample entry is defined as a sample entry on which the following transformation procedure has been applied:

1) The four character code of the sample entry is replaced by a new sample entry code `'resv'` meaning restricted video.

2) A `RestrictedSchemeInfoBox` is added to the sample description, leaving all other boxes unmodified.

3) The original sample entry type is stored within an `OriginalFormatBox` contained in the `RestrictedSchemeInfoBox`.

A `RestrictedSchemeInfoBox` is formatted exactly the same as a `ProtectionSchemeInfoBox`, except that is uses the identifier `'rinf'` instead of `'sinf'` (see below).

The original sample entry type is contained in the `OriginalFormatBox` located in the `RestrictedSchemeInfoBox` (in an identical way to the `ProtectionSchemeInfoBox` for encrypted media).

The exact nature of the restriction is defined in the `SchemeTypeBox`, and the data needed for that scheme is stored in the `SchemeInformationBox`, again, analogously to protection information.

Restriction and protection can be applied at the same time. The order of the transformations follows from the four-character code of the sample entry. For instance, if the sample entry type is `'resv'`, undoing the above transformation may result in a sample entry type `'encv'`, indicating that the media is protected.

If the file author only wants to provide advisory information without stopping legacy players from playing the file, the `RestrictedSchemeInfoBox` may be placed inside the sample entry without transforming the four character code. In this case it is not necessary to include an `OriginalFormatBox`.

8.15.3 Restricted scheme information box

8.15.3.1 Definition

Box Types: `'rinf'`
Container: Restricted Sample Entry or Sample Entry
Mandatory: Yes in Restricted Sample Entry, no otherwise
Quantity: Zero or one

The `RestrictedSchemeInfoBox` contains all the information required both to understand the restriction scheme applied and its parameters. It also documents the original (un-transformed) sample entry type of the media. The `RestrictedSchemeInfoBox` is a container Box. It is mandatory in a sample entry that uses a code indicating a restricted stream, i.e., `'resv'`.

When used in a restricted sample entry, this box shall contain the `OriginalFormatBox` to document the original sample entry type and a `SchemeTypeBox`. A `SchemeInformationBox` may be required depending on the restriction scheme.

8.15.3.2 Syntax

```
aligned(8) class RestrictedSchemeInfoBox(fmt) extends Box('rinf') {
   OriginalFormatBox(fmt)   original_format;
   SchemeTypeBox           scheme_type_box;
   SchemeInformationBox    info;           // optional
}
```

8.15.4 Scheme for stereoscopic video arrangements

8.15.4.1 General

When stereo-coded video frames are decoded, the decoded frames either contain a representation of two spatially packed constituent frames that form a stereo pair (frame packing) or only one view of a stereo pair (left and right views in different tracks). Restrictions due to stereo-coded video are contained in the `StereoVideoBox`.

The SchemeType `'stvi'` (stereoscopic video) is used.

8.15.4.2 Stereo video box

8.15.4.2.1 Definition

Box Type: `'stvi'`
Container: `SchemeInformationBox`
Mandatory: Yes (when the SchemeType is `'stvi'`)
Quantity: One

The `StereoVideoBox` is used to indicate that decoded frames either contain a representation of two spatially packed constituent frames that form a stereo pair or contain one of two views of a stereo pair. The `StereoVideoBox` shall be present when the SchemeType is `'stvi'`.

When the value of `stereo_indication_type` indicates the temporal interleaving frame packing arrangement and the display system in use presents two views simultaneously, readers should implicitly set the composition timestamp for constituent picture 0 to coincide with the composition timestamp for constituent picture 1.

8.15.4.2.2 Syntax

```
aligned(8) class StereoVideoBox extends extends FullBox('stvi', version = 0, 0)
{
   template unsigned int(30) reserved = 0;
   unsigned int(2)    single_view_allowed;
   unsigned int(32)    stereo_scheme;
   unsigned int(32)    length;
   unsigned int(8)[length]    stereo_indication_type;
   Box[] any_box; // optional
}
```

8.15.4.2.3 Semantics

`single_view_allowed` is an integer. A zero value indicates that the content may only be displayed on stereoscopic displays. When (`single_view_allowed` & 1) is equal to 1, it is allowed to display the right view on a monoscopic single-view display. When (`single_view_allowed` & 2) is equal to 2, it is allowed to display the left view on a monoscopic single-view display.

`stereo_scheme` is an integer that indicates the stereo arrangement scheme used and the stereo indication type according to the used scheme. The following values for `stereo_scheme` are specified:

1: the frame packing scheme as specified by the Frame packing arrangement Supplemental Enhancement Information message of ISO/IEC 14496-10:2014,

2: the arrangement type scheme as specified in ISO/IEC 13818-2:2013, Annex D

3: the stereo scheme as specified in ISO/IEC 23000-11[30] for both frame/service compatible and 2D/3D mixed services.

4: a value of **VideoFramePackingType** as defined in ISO/IEC 23091-2.

Other values of `stereo_scheme` are reserved.

`length` indicates the number of bytes for the `stereo_indication_type` field.

`stereo_indication_type` indicates the stereo arrangement type according to the used stereo indication scheme. The syntax and semantics of `stereo_indication_type` depend on the value of `stereo_scheme`. The syntax and semantics for `stereo_indication_type` for the following values of `stereo_scheme` are specified as follows:

`stereo_scheme` equal to 1: The value of `length` shall be 4 and `stereo_indication_type` shall be `unsigned int(32)` which contains the frame_packing_arrangement_type value from ISO/IEC 14496-10:2014, Table D.8 (‘Definition of frame_packing_arrangement_type’).

`stereo_scheme` equal to 2: The value of `length` shall be 4 and `stereo_indication_type` shall be `unsigned int(32)` which contains the type value from ISO/IEC 13818-2:2013, Table D.1 (‘Definition of arrangement_type’).

`stereo_scheme` equal to 3: The value of `length` shall be 2 and `stereo_indication_type` shall contain two syntax elements of `unsigned int(8)`. The first syntax element shall contain the stereoscopic composition type from ISO/IEC 23000-11:2009[30], Table 4. The least significant bit of the second syntax element shall contain the value of `is_left_first` as specified in ISO/IEC 23000-11:2009[30], subclause 8.4.3, while the other bits are reserved and shall be set to 0.

`stereo_scheme` equal to 4: The value of `length` shall be 2 and `stereo_indication_type` shall contain two syntax elements of `unsigned int(8)`. The first syntax element shall contain a **VideoFramePackingType** from ISO/IEC 23091-2. The least significant bit of the second syntax element shall contain the value of **QuincunxSamplingFlag** as specified in ISO/IEC 23091-2, while the other bits are reserved and shall be set to 0. **PackedContentInterpretationType** specified in ISO/IEC 23091-2 is inferred to be equal to 1.

`stereo_scheme` equal to 5: The value of `length` shall be 3 and `stereo_indication_type` shall contain three syntax elements of type `unsigned int(8)`. The first syntax element shall contain a **VideoFramePackingType** from ISO/IEC 23091-2. The least significant bit of the second syntax element shall contain the value of **QuincunxSamplingFlag** as specified in ISO/IEC 23091-2, while the other bits are reserved and shall be set to 0. The third syntax element shall contain the **PackedContentInterpretationType** from ISO/IEC 23091-2.

The following applies when the `StereoVideoBox` is used:

— In the `TrackHeaderBox`

  — `width` and `height` specify the visual presentation size of a single view after unpacking.

— In the `SampleDescriptionBox`

  — `frame_count` shall be 1, because the decoder physically outputs a single frame. In other words, the constituent frames included within a frame-packed picture are not documented by `frame_count`.

  — `width` and `height` document the pixel counts of a frame-packed picture (and not the pixel counts of a single view within a frame-packed picture).

  — the `PixelAspectRatioBox` documents the pixel aspect ratio of each view when the view is displayed on a monoscopic single-view display. For example, in many spatial frame packing arrangements, the pixel aspect ratio box therefore indicates 2:1 or 1:2 pixel aspect ratio, as the spatial resolution of one view of frame-packed video is typically halved along one coordinate axis compared to that of the single-view video of the same format.

8.15.5 Compatible scheme type box

8.15.5.1 Definition

Box Type: `'csch'`
Container: `RestrictedSchemeInfoBox`
Mandatory: No
Quantity: Zero or more

`CompatibleSchemeTypeBox` identifies a scheme type that the track conforms to. The `SchemeTypeBox` and the instances of `CompatibleSchemeTypeBox` provide the possibility to indicate several `scheme_type` values for the same track. The track conforms to all the constraints imposed by all indicated `scheme_type` values among the `SchemeTypeBox` and the instances of `CompatibleSchemeTypeBox` within the same `RestrictedSchemeInfoBox`.

Parsers that are capable of processing any indicated `scheme_type` value among the `SchemeTypeBox` or any instance of `CompatibleSchemeTypeBox` are allowed to process the track provided that they process all the boxes contained in the `SchemeInformationBox`.

8.15.5.2 Syntax

```
aligned(8) class CompatibleSchemeTypeBox extends FullBox('csch', 0, flags) {
   // identical syntax to SchemeTypeBox
   unsigned int(32)   scheme_type;      // 4CC identifying the scheme
   unsigned int(32)   scheme_version;   // scheme version
   if (flags & 0x000001) {
      utf8string scheme_uri;      // browser uri
   }
}
```

8.15.5.3 Semantics

The semantics of the syntax elements are identical to the semantics of the syntax elements with the same name in `SchemeTypeBox`.

8.16 Segments

8.16.1 Overview

Media presentations may be divided into segments for delivery, for example, it is possible (e.g. in HTTP streaming) to form files that contain a segment – or concatenated segments – which would not necessarily form ISO base media file format compliant files (e.g. they do not contain a movie box).

When the MIME form registered in Annex F is used, it shall refer to a segment as defined here.

8.16.2 Segment type box

8.16.2.1 Definition

Box Type: `'styp'`
Container: File
Mandatory: No
Quantity: Zero or more

If segments are stored in separate files (e.g. on a standard HTTP server) it is recommended that these ‘segment files’ contain a segment-type box, which shall be first if present, to enable identification of those files, and declaration of the specifications with which they are compliant.

A `SegmentTypeBox` has the same format as a `FileTypeBox` [4.2.3], except that it takes the box type `'styp'`. The brands within it may include the same brands that were included in the `FileTypeBox` that preceded the `MovieBox`, and may also include additional brands to indicate the compatibility of this segment with various specification(s).

Valid segment type boxes shall be the first box in a segment. Segment type boxes may be removed if segments are concatenated (e.g. to form a full file), but this is not required. Segment type boxes that are not first in their files may be ignored.

8.16.2.2 Syntax

```
aligned(8) class SegmentTypeBox extends GeneralTypeBox ('styp')
{}
```

8.16.3 Segment index box

8.16.3.1 Definition

Box Type: `'sidx'`
Container: File
Mandatory: No
Quantity: Zero or more

The `SegmentIndexBox` provides a compact index of one media stream within the media segment to which it applies. It is designed so that it can be used not only with media formats based on this document (i.e. segments containing sample tables or movie fragments), but also other media formats (for example, MPEG-2 transport streams in ISO/IEC 13818-1[15]). For this reason, the formal description of the box given here is deliberately generic, and then at the end of this subclause the specific definitions for segments using movie fragments are given.

Each `SegmentIndexBox` documents how a (sub)segment is divided into one or more subsegments (which may themselves be further subdivided using `SegmentIndexBox`es).

A subsegment is defined as a time interval of the containing (sub)segment, and corresponds to a single range of bytes of the containing (sub)segment. The durations of all the subsegments sum to the duration of the containing (sub)segment.

Each entry in the `SegmentIndexBox` contains a reference type that indicates whether the reference points directly to the media bytes of a referenced leaf subsegment, or to a `SegmentIndexBox` that describes how the referenced subsegment is further subdivided; as a result, the segment may be indexed in a 'hierarchical' or 'daisy-chain' or other form by documenting time and byte offset information for other `SegmentIndexBox`es applying to portions of the same (sub)segment.

Each `SegmentIndexBox` provides information about a single media stream of the Segment, referred to as the reference stream. If provided, the first `SegmentIndexBox` in a segment, for a given media stream, shall document the entirety of that media stream in the segment, and shall precede any other `SegmentIndexBox` in the segment for the same media stream.

If a segment index is present for at least one media stream but not all media streams in the segment, then normally a media stream in which not every access unit is independently coded, such as video, is selected to be indexed. For any media stream for which no segment index is present, referred to as non-indexed stream, the media stream associated with the first `SegmentIndexBox` in the segment serves as a reference stream in a sense that it also describes the subsegments for any non-indexed media stream.

NOTE 1 Further restrictions can be specified in derived specifications.

`SegmentIndexBox`es may be inline in the same file as the indexed media or, in some cases, in a separate file containing only indexing information.

A `SegmentIndexBox` contains a sequence of references to subsegments of the (sub)segment documented by the box. The referenced subsegments are contiguous in presentation time. Similarly, the bytes referred to by a `SegmentIndexBox` are always contiguous in both the media file, and the separate index segment, or in the single file if indexes are placed within the media file. The referenced size gives the count of the number of bytes in the material referenced.

NOTE 2 A media segment can be indexed by more than one "top-level" `SegmentIndexBox` that are independent of each other, each of which indexes one media stream within the media segment. In segments containing multiple media streams the referenced bytes can contain media from multiple streams, even though the `SegmentIndexBox` provides timing information for only one media stream.

In the file containing the `SegmentIndexBox`, the anchor point for a `SegmentIndexBox` is the first byte after that box. If there are two files, the anchor point in the media file is the beginning of the top-level segment (i.e. the beginning of the segment file if each segment is stored in a separate file). The material in the file containing media (which may also be the file that contains the `SegmentIndexBox`es) starts at the indicated offset from the anchor point. If there are two files, the material in the index file starts at the anchor point, i.e. immediately following the `SegmentIndexBox`.

Within the two constraints (a) that, in time, the subsegments are contiguous, that is, each entry in the loop is consecutive from the immediately preceding one and (b) within a given file (integrated file, media file, or separate file containing only index information) the referenced bytes are contiguous, there are a number of possibilities, including:

1) a reference to a `SegmentIndexBox` may include, in its byte count, immediately following `SegmentIndexBox`es that document subsegments;

2) in an integrated file, using the `first_offset` field, it is possible to separate `SegmentIndexBox`es from the media that they refer to;

3) in an integrated file, it is possible to locate `SegmentIndexBox`es for subsegments close to the media they index;

4) when a separate file containing `SegmentIndexBox`es is used, it is possible for the loop entries to have different `reference_type` values, some to `SegmentIndexBox`es in the index segment, some to media subsegments in the media file.

NOTE 3 Profiles can be used to restrict the placement of segment indexes, or the overall complexity of the indexing.

The `SegmentIndexBox` documents the presence of stream access points (SAPs), as specified in Annex I, in the referenced subsegments. The annex specifies characteristics of SAPs, such as I_{SAU}, I_{SAP} and T_{SAP}, as well as SAP types, which are all used in the semantics below. A subsegment starts with a SAP when the subsegment contains a SAP, and for the first SAP, I_{SAU} is the index of the first access unit that follows I_{SAP}, and I_{SAP} is contained in the subsegment.

For segments based on this document (i.e. based on movie sample tables or movie fragments):

— an access unit is a sample;

— a subsegment is a self-contained set of one or more consecutive movie fragments; a self-contained set contains one or more `MovieFragmentBox`es with the corresponding `MediaDataBox`(es), and a `MediaDataBox` containing data referenced by a `MovieFragmentBox` shall follow that `MovieFragmentBox` and precede the next `MovieFragmentBox` containing information about the same track;

— Segment index boxes shall be placed before subsegment material they document, that is, before any `MovieFragmentBox` of the documented material of the subsegment;

— streams are tracks in the file format, and stream IDs are `track_ID`s;

— a subsegment contains a stream access point if a track fragment within the subsegment for the track with `track_ID` equal to `reference_ID` contains a stream access point;

— initialisation data for SAPs consists of the movie box;

— presentation times are in the presentation timeline, that is they are composition times after the application of any edit list for the track;

— the I_{SAP} is a position exactly pointing to the start of a top-level box, such as a movie fragment box `'moof'`;

— a SAP of type 1 or type 2 is indicated as a sync sample, or by `sample_is_non_sync_sample` equal to 0 in the movie fragment;

— a SAP of type 3 is marked as a member of a sample group of type `'rap '`;

— a SAP of type 4 is marked as a member of a sample group of type `'roll'` where the value of the `roll_distance` field is greater than 0.

NOTE 4 For SAPs of type 5 and 6, no specific signalling in the ISO base media file format is supported.

Examples of the use of segment indexes can be found in Annex J.

8.16.3.2 Syntax

```
aligned(8) class SegmentIndexBox extends FullBox('sidx', version, 0) {
   unsigned int(32) reference_ID;
   unsigned int(32) timescale;
   if (version==0) {
         unsigned int(32) earliest_presentation_time;
         unsigned int(32) first_offset;
      }
      else {
         unsigned int(64) earliest_presentation_time;
         unsigned int(64) first_offset;
      }
   unsigned int(16) reserved = 0;
   unsigned int(16) reference_count;
   for(i=1; i <= reference_count; i++)
   {
      bit (1)             reference_type;
      unsigned int(31)    referenced_size;
      unsigned int(32)    subsegment_duration;
      bit(1)             starts_with_SAP;
      unsigned int(3)    SAP_type;
      unsigned int(28)    SAP_delta_time;
   }
}
```

8.16.3.3 Semantics

`reference_ID` provides the stream ID for the reference stream; if this `SegmentIndexBox` is referenced from a “parent” `SegmentIndexBox`, the value of `reference_ID` shall be the same as the value of `reference_ID` of the “parent” `SegmentIndexBox`;

`timescale` provides the timescale, in ticks per second, for the time and duration fields within this box; it is recommended that this match the timescale of the reference stream or track; for files based on this document, that is the timescale field of the media header box of the track;

`earliest_presentation_time` is the earliest presentation time of any content in the reference stream in the first subsegment, in the timescale indicated in the timescale field; the earliest presentation time is derived from media in access units, or parts of access units, that are not omitted by an edit list (if any);

`first_offset` is the distance in bytes, in the file containing media, from the anchor point, to the first byte of the indexed material;

`reference_count` provides the number of referenced items;

reference_type: when set to 1 indicates that the reference is to a SegmentIndexBox; otherwise the reference is to media content (e.g., in the case of files based on this document, to a MovieFragmentBox); if a separate index segment is used, then entries with reference type 1 are in the index segment, and entries with reference type 0 are in the media file;

referenced_size: the distance in bytes from the first byte of the referenced item to the first byte of the next referenced item, or in the case of the last entry, the end of the referenced material;

subsegment_duration: when the reference is to SegmentIndexBox, this field carries the sum of the subsegment_duration fields in that box; when the reference is to a subsegment, this field carries the difference between the earliest presentation time of any access unit of the reference stream in the next subsegment (or the first subsegment of the next segment, if this is the last subsegment of the segment, or the end presentation time of the reference stream if this is the last subsegment of the stream) and the earliest presentation time of any access unit of the reference stream in the referenced subsegment; the duration is in the same units as earliest_presentation_time;

starts_with_SAP indicates whether the referenced subsegments start with a SAP. For the detailed semantics of this field in combination with other fields, see Table 6.

SAP_type indicates a SAP type as specified in Annex I, or the value 0. Other type values are reserved. For the detailed semantics of this field in combination with other fields, see the table below.

SAP_delta_time: indicates T_{SAP} of the first SAP, in decoding order, in the referenced subsegment for the reference stream. If the referenced subsegments do not contain a SAP, SAP_delta_time is reserved with the value 0; otherwise SAP_delta_time is the difference between the earliest presentation time of the subsegment, and the T_{SAP} (this difference may be zero, in the case that the subsegment starts with a SAP).

Table 6 — Semantics of SAP and reference type combinations

starts_with_SAP	SAP_type	reference_type	Meaning
0	0	0 or 1	No information of SAPs is provided.
0	1 to 6, inclusive	0 (media)	The subsegment contains (but may not start with) a SAP of the given SAP_type and the first SAP of the given SAP_type corresponds to SAP_delta_time.
0	1 to 6, inclusive	1 (index)	All the referenced subsegments contain a SAP of at most the given SAP_type and none of these SAPs is of an unknown type.
1	0	0 (media)	The subsegment starts with a SAP of an unknown type.
1	0	1 (index)	All the referenced subsegments start with a SAP which may be of an unknown type
1	1 to 6, inclusive	0 (media)	The referenced subsegment starts with a SAP of the given SAP_type.
1	1 to 6, inclusive	1 (index)	All the referenced subsegments start with a SAP of at most the given SAP_type and none of these SAPs is of an unknown type.

8.16.4 Subsegment index box

8.16.4.1 Definition

Box Type: 'ssix'
Container: File
Mandatory: No
Quantity: Zero or more

The SubsegmentIndexBox provides a mapping from levels (as specified by the LevelAssignmentBox) to byte ranges of the indexed subsegment. In other words, this box provides a compact index for how the data in a subsegment is ordered according to levels into partial subsegments. It enables a client to easily access data for partial subsegments by downloading ranges of data in the subsegment.

Each byte in the subsegment shall be explicitly assigned to a level, and hence the range count shall be 2 or greater. If the range is not associated with any information in the level assignment, then any level that is not included in the level assignment may be used.

There shall be 0 or 1 SubsegmentIndexBoxes per each SegmentIndexBox that indexes only leaf subsegments, i.e. that only indexes subsegments but no segment indexes. A SubsegmentIndexBox, if any, shall be the next box after the associated SegmentIndexBox. A SubsegmentIndexBox documents the subsegments that are indicated in the immediately preceding SegmentIndexBox.

In general, the media data constructed from the byte ranges is incomplete, i.e. it does not conform to the media format of the entire subsegment.

For leaf subsegments based on this document (i.e. based on movie sample tables and movie fragments):

— Each level shall be assigned to exactly one partial subsegment, i.e. byte ranges for one level shall be contiguous.

— Levels of partial subsegments shall be assigned by increasing numbers within a subsegment, i.e., samples of a partial subsegment may depend on any samples of preceding partial subsegments in the same subsegment, but not the other way around. For example, each partial subsegment contains samples having an identical temporal level and partial subsegments appear in increasing temporal level order within the subsegment.

— When a partial subsegment is accessed in this way, for any assignment_type other than 3, the final MediaDataBox may be incomplete, that is, less data is accessed than the length indication of the MediaDataBox indicates is present. The length of the MediaDataBox may need adjusting, or padding used. The padding_flag in the LevelAssignmentBox indicates whether this missing data can be replaced by zeros. If not, the sample data for samples assigned to levels that are not accessed is not present, and care should be taken not to attempt to process such samples.

— The data ranges corresponding to partial subsegments include both MovieFragmentBoxes and MediaDataBoxes. The first partial subsegment, i.e. the lowest level, will correspond to a MovieFragmentBox as well as (parts of) MediaDataBox(es), whereas subsequent partial subsegments (higher levels) may correspond to (parts of) MediaDataBox(es) only.

NOTE assignment_type equal to 0 (specified in the LevelAssignmentBox) can be used, for example, together with the temporal level sample grouping ('tele') when frames of a video bitstream are temporally ordered within subsegments; assignment_type equal to 2 can be used, for example, when each view of a multiview video bitstream is contained in a separate track and the track fragments for all the views are contained in a single movie fragment. assignment_type equal to 3 can be used, for example, when audio and video movie fragments (including the respective MediaDataBoxes) are interleaved. The first level can be specified to contain the audio movie fragments (including the respective MediaDataBoxes), whereas the second level can be specified to contain both audio and video movie fragments (including all MediaDataBoxes).

8.16.4.2 Syntax

```
aligned(8) class SubsegmentIndexBox extends FullBox('ssix', 0, 0) {
   unsigned int(32)   subsegment_count;
   for( i=1; i <= subsegment_count; i++)
   {
      unsigned int(32)   range_count;
      for ( j=1; j <= range_count; j++) {
         unsigned int(8) level;
         unsigned int(24) range_size;
      }
   }
}
```

8.16.4.3 Semantics

subsegment_count is a positive integer specifying the number of subsegments for which partial subsegment information is specified in this box. subsegment_count shall be equal to reference_count (i.e., the number of movie fragment references) in the immediately preceding SegmentIndexBox.

range_count specifies the number of partial subsegment levels into which the media data is grouped. This value shall be greater than or equal to 2.

range_size indicates the size of the partial subsegment; the value 0 may be used in the last entry to indicate the remaining bytes of the segment, to the end of the segment.

level specifies the level to which this partial subsegment is assigned.

8.16.5 Producer reference time box

8.16.5.1 Definition

Box Type: 'prft'
Container: File
Mandatory: No
Quantity: Zero or more

The ProducerReferenceTimeBox supplies times corresponding to the production of associated movie fragments. When they are both produced and consumed in real time, this can provide clients with information to enable consumption and production to proceed at equivalent rates, thus avoiding possible buffer overflow or underflow.

This box is related to the next MovieFragmentBox that follows it in bitstream order. It shall follow any SegmentTypeBox or SegmentIndexBox (if any) in the segment, and occur before the following MovieFragmentBox (to which it refers).

The box contains a time value, expressed using the NTP format, and measured on a clock which increments at the same rate as a UTC-synchronized NTP clock as defined in IETF RFC 5905. This time is associated with a media time in the range of media times for one of the tracks in the movie fragment. Note that the media time does not need to match the actual decode or composition timestamp of a sample of that track.

Given a succession of ProducerReferenceTimeBoxes in the same track with the same flag values (notably the value 8, which promises consistency), a receiver can estimate the drift between the producer's media clock and real time; the accuracy and variability of this estimation can depend on the technique used to establish the times, as reported in the flag values.

For live-captured media, flags may be set to 24 (i.e. the two bits corresponding to value 8 and 16 are set). When the flags have this value, the UTC time shall be at, or close to, the wall-clock-time of the experience captured in the media. For example, if the media is a video that shows an accurate public clock timed to UTC that displays 09:15, then the UTC time in this box would also read close to 09:15. The receiver can then estimate the real-time latency of presentation, from the time the captured experience occurred, to the time it is presented to the user.

Producer reference times should be associated with at most one track.

8.16.5.2 Syntax

```
aligned(8) class ProducerReferenceTimeBox
   extends FullBox('prft', version, flags) {
   unsigned int(32) reference_track_ID;
   unsigned int(64) ntp_timestamp;
   if (version==0) {
      unsigned int(32) media_time;
   } else {
      unsigned int(64) media_time;
   }
}
```

8.16.5.3 Semantics

reference_track_ID provides the track_ID for the reference track.

ntp_timestamp indicates a UTC time in NTP format associated to media_time as follows:

— if flags is set to 0, the UTC time is the time at which the frame belonging to the reference track in the following movie fragment and whose presentation time is media_time was input to the encoder.

— if flags is set to 1, the UTC time is the time at which the frame belonging to the reference track in the following movie fragment and whose presentation time is media_time was output from the encoder.

— if flags is set to 2, the UTC time is the time at which the following MovieFragmentBox was finalized. media _time is set to the presentation of the earliest frame of the reference track in presentation order of the movie fragment.

— if flags is set to 4, the UTC time is the time at which the following MovieFragmentBox was written to file. media _time is set to the presentation of the earliest frame of the reference track in presentation order of the movie fragment.

— if flags is set to 8, the association between the media_time and UTC time is arbitrary but consistent between multiple occurrences of this box in the same track

— if flags is set to 24 (i.e. the two bits corresponding to value 8 and 16 are set), the UTC time has a consistent, small (ideally zero), offset from the real-time of the experience depicted in the media at media_time

media_time is expressed in the time units used for the reference track.

NOTE In most cases this time will not be equal to the time of the first sample of the adjacent segment of the reference track.

8.17 Support for incomplete tracks

8.17.1 General

This subclause documents the sample entry formats for tracks that are incomplete. Incomplete tracks may contain samples that are marked empty or not received using the sample format.

Incomplete tracks may result, for example, when subsegments are received partially according to level assignments and padding_flag in the LevelAssignmentBox indicates that the data in a MediaDataBox that is not received can be replaced by zeros. Consequently, sample data assigned to non-accessed levels is not present, and care should be taken not to attempt to process such samples. However, in partially received subsegments some tracks might remain complete in content while other tracks might be incomplete and only contain data that is included by reference into the complete tracks.

This subclause specifies support for sample entry formats for incomplete tracks. With this support, readers can detect incomplete tracks from their sample entries and avoid processing such tracks or take the possibility of empty or not received samples into account when processing such tracks.

The support for incomplete tracks is similar to the content protection transformation where sample entries are hidden behind generic sample entries, such as `'encv'` and `'enca'`. Because the format of a sample entry varies with media-type, a different encapsulating four character code is used for incomplete tracks of each media type (audio, video, text etc.). They are:

Stream (Track) Type	**Sample-Entry Code**
Video	`icpv`
Audio	`icpa`
Text	`icpt`
System	`icps`
Hint	`icph`
Haptics	`icpp`
Volumetric visual	`icp3`
Timed Metadata	`icpm`

Sample data of incomplete tracks may be included into samples of other tracks by reference, and hence an incomplete track should not be removed as long as any track reference points to it.

NOTE 1 The choice of level by the original recording client can vary over time, and at times represent the complete track. The level is not indicated here, and it is not required that the sample entry change from ‘incomplete’ to ‘complete’ when all levels were, in fact, received, for a period.

NOTE 2 The ‘original format’ might have indicated encryption, if partial reception and decryption works for that encryption format.

8.17.2 Transformation

The sample entry for a track that becomes incomplete e.g. through partial reception, should be modified as follows:

1) The four character code of the sample entry, e.g. `'avc1'`, is replaced by a new sample entry code `'icpv'` meaning an incomplete track.

2) A `CompleteTrackInfoBox` is added to the sample description, leaving all other boxes unmodified.

3) The original sample entry type, e.g. `'avc1'`, is stored within an `OriginalFormatBox` contained in the `CompleteTrackInfoBox`.

After transformation, an example AVC sample entry might look like:

```
class IncompleteAVCSampleEntry() extends VisualSampleEntry ('icpv'){
   CompleteTrackInfoBox();
   AVCConfigurationBox config;
}
```

NOTE The sample entry type `'avc1'` and the `AVCConfigurationBox` are specified in ISO/IEC 14496-15[16].

8.17.3 Complete track information box

8.17.3.1 Definition

Box Types: `'cinf'`
Container: Sample Entry for an Incomplete Track
Mandatory: Yes
Quantity: Exactly one

The `CompleteTrackInfoBox` contains, within the `OriginalFormatBox`, the sample entry format of the complete track that was transformed to the present incomplete track. It may contain optional boxes for example including information required to process samples of the present incomplete track. The

CompleteTrackInfoBox is a container box. It is mandatory in a sample entry that uses a code indicating an incomplete track.

8.17.3.2 Syntax

```
aligned(8) class CompleteTrackInfoBox(fmt) extends Box('cinf') {
   OriginalFormatBox(fmt)   original_format;
}
```

8.18 Entity grouping

8.18.1 General

An entity group is a grouping of items, which may also group tracks. The entities in an entity group share a particular characteristic or have a particular relationship, as indicated by the grouping type.

Entity groups are indicated in GroupsListBox. Entity groups specified in GroupsListBox of a file-level MetaBox refer to tracks or file-level items. Entity groups specified in GroupsListBox of a movie-level MetaBox refer to movie-level items. Entity groups specified in GroupsListBox of a track-level MetaBox refer to track-level items of that track.

GroupsListBox contains EntityToGroupBoxes, each specifying one entity group.

8.18.2 Groups list box

8.18.2.1 Definition

Box Type: 'grpl'
Container: MetaBox
Mandatory: No
Quantity: Zero or One

The GroupsListBox includes the entity groups specified for the file. This box contains a set of full boxes, each called an EntityToGroupBox, with four-character codes denoting a defined grouping type.

When GroupsListBox is present in a file-level MetaBox, there shall be no item_ID value in ItemInfoBox in any file-level MetaBox that is equal to the track_ID value in any TrackHeaderBox.

8.18.2.2 Syntax

```
aligned(8) class GroupsListBox extends Box('grpl') {
}
```

8.18.3 Entity to group box

8.18.3.1 Definition

Box Type: As specified below with the grouping_type value for the EntityToGroupBox
Container: GroupsListBox
Mandatory: No
Quantity: One or more

The EntityToGroupBox specifies an entity group.

The box type (grouping_type) indicates the grouping type of the entity group. Each grouping_type code is associated with semantics that describe the grouping. The following grouping_type value is specified:

'altr': The items and tracks mapped to this grouping are alternatives to each other, and only one of them should be played (when the mapped items and tracks are part of the presentation; e.g. are displayable items or tracks) or processed by other means (when the mapped items or tracks are not part of the presentation; e.g. are metadata). A player should select the first entity from the list of entity_id values that it can process (e.g. decode and play for mapped items and tracks that are part of the presentation) and that suits the application needs. Any entity_id value shall be mapped to only one grouping of type 'altr'. An alternate group of entities consists of those items and tracks that are mapped to the same entity group of type 'altr'.

NOTE EntityToGroupBox can have grouping_type specific extensions.

8.18.3.2 Syntax

```
aligned(8) class EntityToGroupBox(grouping_type, version, flags)
extends FullBox(grouping_type, version, flags) {
   unsigned int(32) group_id;
   unsigned int(32) num_entities_in_group;
   for(i=0; i<num_entities_in_group; i++)
      unsigned int(32) entity_id;
// the remaining data may be specified for a particular grouping_type
}
```

8.18.3.3 Semantics

group_id is a non-negative integer assigned to the particular grouping that shall not be equal to any group_id value of any other EntityToGroupBox, any item_ID value of the hierarchy level (file, movie. or track) that contains the GroupsListBox, or any track_ID value (when the GroupsListBox is contained in the file level).

num_entities_in_group specifies the number of entity_id values mapped to this entity group.

entity_id is resolved to an item, when an item with item_ID equal to entity_id is present in the hierarchy level (file, movie or track) that contains the GroupsListBox, or to a track, when a track with track_ID equal to entity_id is present and the GroupsListBox is contained in the file level.

8.19 Compressed boxes

8.19.1 Overview and processing

A compressed box is a box that replaces another box, called original box, at the same top-level order in the file but whose payload is the BoxPayload of that original box, compressed with deflate(), as defined in IETF RFC 1951. A compressed box can only be defined for a top-level box; hence their container is always the file and is omitted in their box definitions. Only a subset of existing top-level boxes can be compressed.

The type of the box that a compressed box replaces is called the replacement type and is defined for each compressed box. The content that is compressed comprises all the remaining bytes after the BoxHeader of the original box.

Unless stated otherwise, the semantics of fields in the uncompressed payload is the same as the semantics of the box with the given replacement type. Especially, when boxes use file offsets within the containing file, unless stated otherwise these offsets are related to the file that results from expanding the compressed box(es).

All compressed boxes defined in this document share the same syntax as defined in subclause 8.19.3.

The MovieFragmentRandomAccessBox is not supported in files containing compressed boxes and shall not be present when a compressed box is present.

A reader that supports compressed boxes shall be capable of decompressing boxes and adjusting offsets in a way that is functionally equivalent to the processing model in subclause 8.19.2.2.

8.19.2 Processing model

8.19.2.1 File parsing processing model

The processing model when reading compressed boxes is defined as follows:

— the `BoxPayload` of the compressed box (after the compressed box's `BoxHeader` structure) is decompressed (note that a `FullBox` also has a `BoxPayload`)

— the box type of the compressed box is replaced with the replacement type as specified for this particular compressed box

— the compressed box size is replaced with the sum of the uncompressed payload size and the `BoxHeader` size (8 bytes, in the case of a simple 32-bit size)

— the compressed box payload is replaced with the uncompressed payload

— the uncompressed box is read from the reconstructed `BoxHeader` and uncompressed payload

— the parsing is then resumed at the position of the end of the compressed box in the file.

8.19.2.2 File decompression processing model

The processing model when decompressing a file is defined as follows: the decompressed file is initialized as empty, and for each top-level box in the compressed file:

— if there is an OriginalFileTypeBox, then the FileTypeBox or SegmentTypeBox, and a following `ExtendedTypeBox` (if any), are replaced by the contents of the OriginalFileTypeBox;

— otherwise, if the box is uncompressed, it is copied to the decompressed file.

— otherwise, the decompressed box is computed as indicated in subclause 8.19.2.1; the decompressed box is written to the decompressed file as follows:

  — if the decompressed box is a MovieBox or MovieFragmentBox, the decompressed box is copied without modifications to the decompressed file.

  — if the decompressed box is a SegmentIndexBox

    — the first_offset field is incremented by the number of extra bytes in the decompressed data between the beginning of the SegmentIndexBox and the indicated first_offset in the compressed file, compared to the number of compressed bytes in that range; (this necessarily includes the expansion of the SegmentIndexBox itself); this step is needed when SegmentIndexBox(es) and SubSegmentIndexBox(es) are present between the end of an SegmentIndexBox and the first movie fragment following this SegmentIndexBox.

    — Each reference_size field in the decompressed SegmentIndexBox is increased by the number of uncompressed bytes minus the number of compressed bytes of all top-level compressed boxes present in the corresponding range in the compressed file. (In general it is not possible to update a `SegmentIndexBox` immediately on reception, as the first_offset and reference_size are updated based on decompression of boxes later in the file.)

    — The modified uncompressed box is then copied to the decompressed file.

  — if the decompressed box is a SubsegmentIndexBox, each range_size field in the uncompressed box is increased by the number of uncompressed bytes minus the number of compressed bytes of all top-level compressed boxes starting in the corresponding range in the compressed file. The modified uncompressed box is then copied to the decompressed file.

NOTE 1 There is no guarantee that the number of compressed bytes is less than the number of uncompressed bytes, although it is better that file writers avoid using a compressed box if it is larger than the uncompressed version. In the above processing model, this means that the number of bytes to add might be negative.

NOTE 2 In this processing mode, a file decompressor might need to load top-level boxes following the box currently being decompressed (e.g., SegmentIndexBox or SubsegmentIndexBox) to compute the byte difference between compressed and uncompressed versions of subsequent boxes referred to by the box currently being decompressed.

NOTE 3 Since, unless specified otherwise, byte offsets in compressed boxes describe the uncompressed domain file, a file reader processing a compressed file on-the-fly (without producing a complete uncompressed file) will need to maintain some correspondence table between byte offsets in the compressed domain and byte offsets in the uncompressed domain in order to correctly adjust file offsets when fetching data from the file. File offsets can only be interpreted once the reader is sure that all data preceding that file offset has been decompressed. This can be achieved by either (1) loading all that data and decompressing it or (2) knowing that the only boxes that precede the given offset, that are permitted to be compressed, have been decompressed. For example, in many cases a file is only permitted to contain only a single MovieBox, a MovieFragmentBox or a MetaBox and no other compressible box. Derived specification can further restrict the possible set and layout of compressed boxes to simplify this process.

NOTE 4 This is a processing model that describes the correspondence between a file containing compressed boxes and a hypothetical file containing uncompressed boxes; readers implement decompression as suits their processing needs. In particular, if SegmentIndexBoxes are used to guide what data to fetch and then not used further, it might not be necessary to recalculate the offsets and sizes that they contain.

8.19.3 General syntax

```
aligned(8) class CompressedBox(box_type, replacement_type)
   extends Box(box_type) {
      bit(8) compressed_data[];// to end of box
}
```

8.19.4 General semantics

compressed_data is the compressed payload of a box defined by the replacement_type type. This data contains the compressed box payload excluding the BoxHeader field of the uncompressed box.

replacement_type indicates the corresponding uncompressed box type.

8.19.5 Original file-type box

8.19.5.1 Definition

Box Type: 'otyp'
Container: File or OriginalFileTypeBox
Mandatory: Yes (see below)
Quantity: Zero or More

A file conformant to this specification may have further transformation of its box structure, for example it may use compressed top-level boxes. In this case, the resulting file may no longer be compliant with the brand promises of the original file, as it requires the support for new tools (such as compressed top-level boxes). For example, a file using a compressed MovieBox is no longer compliant to any brand defined prior to the introduction of compressed boxes.

The OriginalFileTypeBox is used to encapsulate brand information applying to the original file before transformation but not valid in the transformed domain. There shall be at most one OriginalFileTypeBox after each FileTypeBox or SegmentTypeBox. An OriginalFileTypeBox shall contain the FileTypeBox and, when present, the ExtendedTypeBox, of the file before transformation. In cases where a file uses multiple transformations, nested OriginalFileTypeBox shall be used, and there shall be at most one OriginalFileTypeBox in each OriginalFileTypeBox.

When present, an OriginalFileTypeBox shall follow a FileTypeBox or SegmentTypeBox, with at most an ExtendedTypeBox and/or FreeSpaceBox(es) in between.

The processing model for a file reader is equivalent to removing both FileTypeBox/SegmentTypeBox and OriginalFileTypeBox, and inserting in their place the child boxes of the OriginalFileTypeBox. In the case of compressed top-level boxes, the resulting replacement and decompression of the file shall be a compliant uncompressed ISOBMFF file.

8.19.5.2 Syntax

```
aligned(8) class OriginalFileTypeBox extends Box('otyp') {
}
```

8.19.6 Compressed movie box

8.19.6.1 Definition

Box Type: `'!mov'`
Replacement Type: `'moov'`
Mandatory: No
Quantity: Zero or One

A `CompressedMovieBox` contains a compressed version of a `MovieBox BoxPayload`. The replacement type of the `CompressedMovieBox` for the algorithm specified in subclause 8.19.1 is `'moov'`.

There shall not be both a `CompressedMovieBox` and a `MovieBox` in a file.

8.19.6.2 Syntax

```
aligned(8) class CompressedMovieBox
   extends CompressedBox('!mov', 'moov') {
}
```

8.19.7 Compressed movie fragment box

8.19.7.1 Definition

Box Type: `'!mof'`
Replacement Type: `'moof'`
Mandatory: No
Quantity: Zero or One

A `CompressedMovieFragmentBox` contains a compressed version of a `MovieFragmentBox BoxPayload`. The replacement type of the `CompressedMovieFragmentBox` for the algorithm specified in subclause 8.19.1 is `'moof'`.

8.19.7.2 Syntax

```
aligned(8) class CompressedMovieFragmentBox
   extends CompressedBox('!mof', 'moof') {
}
```

8.19.8 Compressed segment index box

8.19.8.1 Definition

Box Type: `'!six'`
Replacement Type: `'sidx'`
Mandatory: No
Quantity: Zero or More

A `CompressedSegmentIndexBox` contains a compressed version of a `SegmentIndexBox BoxPayload`. The replacement type of the `CompressedSegmentIndexBox` for the algorithm specified in subclause 8.19.1 is `'sidx'`.

The `referenced_size` and `first_offset` fields of the uncompressed `SegmentIndexBox` payload shall be expressed in the compressed domain: they document the size and position of the referenced item in the file, and the referenced items may contain compressed top-level boxes. This ensures that byte-ranges computed from the referenced_size fields always resolve in one or more top-level boxes of the file, potentially using box compression.

If a physical version of the decompressed file is to be produced for later consumption, the referenced_size and first_offset fields in the uncompressed payload may need to be updated according to the new size of the referenced items after decompression.

If the SegmentIndexBox references an external file using box compression the referenced_size and first_offset fields shall also be expressed in the compressed domain for the same reason as stated above.

8.19.8.2 Syntax

```
aligned(8) class CompressedSegmentIndexBox
   extends CompressedBox('!six', 'sidx') {
}
```

8.19.9 Compressed subsegment index box

8.19.9.1 Definition

Box Type: '!ssx'
Replacement Type: 'ssix'
Mandatory: No
Quantity: Zero or More

A CompressedSubsegmentIndexBox contains a compressed version of a SubsegmentIndexBox BoxPayload. The replacement type of the CompressedSubsegmentIndexBox for the algorithm specified in subclause 8.19.1 is 'ssix'.

The range_size fields of the uncompressed SubsegmentIndexBox payload shall be expressed in the compressed domain: it documents the size of the partial subsegment in the subsegment, potentially containing compressed top-level boxes. This ensures that byte-ranges computed from the range_size fields always resolve in one or more top-level boxes of the file, potentially using box compression, or in a subset of a MediaDataBox payload.

8.19.9.2 Syntax

```
aligned(8) class CompressedSubsegmentIndexBox
   extends CompressedBox('!ssx', 'ssix') {
}
```

9 Hint track formats

9.1 RTP and SRTP hint track format

9.1.1 Overview

This subclause defines a hint track format for RTP; when processed by a server, the hint track shall yield a stream of RTP packets that conform to IETF RFC 3550[9] and IETF RFC 3551[10].

In standard RTP, each media stream is sent as a separate RTP stream; multiplexing is achieved by using IP's port-level multiplexing, not by interleaving the data from multiple streams into a single RTP session. However, if MPEG is used, it may be necessary to multiplex several media tracks into one RTP track (e.g. when using MPEG-2 transport in RTP, or FlexMux). Each hint track is therefore tied to a set of media tracks by track references. The hint tracks extract data from their media tracks by indexing through this table. Hint track references to media tracks have the reference type 'hint'.

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 `UserDataBox`es in the track.

RTP hint tracks do not use the `CompositionOffsetBox`. Instead, the hinting process for server hint tracks establishes the correct transmission order and timestamps, perhaps using the transmission time offset to set transmission times.

Hinted content may require the use of SRTP for streaming by using the hint track format for SRTP, defined here. SRTP hint tracks shall form a stream of packets that shall conform to IETF RFC 3711, and are formatted identically to RTP hint tracks, except that:

1) the sample entry name is changed from `'rtp '` to `'srtp'` to indicate to the server that SRTP is required;

2) an extra box is added to the sample entry which can be used to instruct the server in the nature of the on-the-fly encryption and integrity protection that must be applied.

9.1.2 Sample description format

9.1.2.1 Structure

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

```
class RtpHintSampleEntry() extends HintSampleEntry ('rtp ') {
   uint(16)      hinttrackversion = 1;
   uint(16)      highestcompatibleversion = 1;
   uint(32)      maxpacketsize;
}
```

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

The `maxpacketsize` indicates the size of the largest packet that this track will generate.

The additional data is a set of boxes, from the following.

```
class timescaleentry() extends Box('tims') {
   uint(32)   timescale;
}

class timeoffset() extends Box('tsro') {
   int(32)      offset;
}

class sequenceoffset() extends Box('snro') {
   int(32)      offset;
}
```

The timescale entry is required. The other two are optional. The offsets over-ride the default server behaviour, which is to choose a random offset. A value of 0, therefore, will cause the server to apply no offset to the timestamp or sequence number respectively.

An SRTP Hint Sample entry is used when it is required that SRTP processing is required.

```
class SrtpHintSampleEntry() extends HintSampleEntry ('srtp') {
   uint(16)      hinttrackversion = 1;
   uint(16)      highestcompatibleversion = 1;
   uint(32)      maxpacketsize;
}
```

Fields and boxes are defined as for the `RtpHintSampleEntry` (`'rtp '`) of the ISO Base Media File Format. However, an `SRTPProcessBox` shall be included in an `SrtpHintSampleEntry` as one of the `additionaldata` boxes.

9.1.2.2 SRTP process box

Box Type: `'srpp'`
Container: SrtpHintSampleEntry
Mandatory: Yes
Quantity: Exactly one

The `SRTPProcessBox` may instruct the server as to which SRTP algorithms should be applied.

```
aligned(8) class SRTPProcessBox extends FullBox('srpp', version, 0) {
   unsigned int(32)      encryption_algorithm_rtp;
   unsigned int(32)      encryption_algorithm_rtcp;
   unsigned int(32)      integrity_algorithm_rtp;
   unsigned int(32)      integrity_algorithm_rtcp;
   SchemeTypeBox         scheme_type_box;
   SchemeInformationBox   info;
}
```

The `SchemeTypeBox` and `SchemeInformationBox` have the syntax defined above for protected media tracks. They serve to provide the parameters required for applying SRTP. The `SchemeTypeBox` is used to indicate the necessary key-management and security policy for the stream in extension to the defined algorithmic pointers provided by the `SRTPProcessBox`. The key-management functionality is also used to establish all the necessary SRTP parameters as listed in IETF RFC 3711:2004, Section 8.2. The exact definition of protection schemes is out of the scope of the file format.

The algorithms for encryption and integrity protection are defined by SRTP. The following format identifiers are defined here. An entry of four spaces (`$20$20$20$20`) may be used to indicate that the choice of algorithm for either encryption or integrity protection is decided by a process outside the file format.

Format	Algorithm
`$20$20$20$20`	The choice of algorithm for either encryption or integrity protection is decided by a process outside the file format
`ACM1`	Encryption using AES in Counter Mode with 128-bit key, as defined in IETF RFC 3711:2004, Section 4.1.1.
`AF81`	Encryption using AES in F8-mode with 128-bit key, as defined in IETF RFC 3711:2004, Section 4.1.2.
`ENUL`	Encryption using the NULL-algorithm as defined in IETF RFC 3711:2004, Section 4.1.3.
`SHM2`	Integrity protection using HMAC-SHA-1 with 160-bit key, as defined in IETF RFC 3711:2004, Section 4.2.1.
`ANUL`	Integrity protection not applied to RTP (but still applied to RTCP). NOTE This is valid only for `integrity_algorithm_rtp`.

9.1.3 Sample format

9.1.3.1 Sample format definition

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.

Each sample contains two areas: the instructions to compose the packets, and any extra data needed when sending those packets (e.g. an encrypted version of the media data). The size of the sample is known from the sample size table.

```
aligned(8) class RTPsample {
   unsigned int(16)   packetcount;
   unsigned int(16)   reserved;
   RTPpacket   packets[packetcount];
   byte      extradata[];
}
```

9.1.3.2 Packet entry format

Each packet in the packet entry table has the following structure:

```
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];
}
```

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.

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

The following 2 bytes exactly overlay the RTP header; they assist the server in making the RTP header (the server fills in the remaining fields). 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.

The sequence seed is the basis for the RTP sequence number. If a hint track causes multiple copies of the same RTP packet to be sent, then the seed value would be the same for them all. The server normally adds a random offset to this value (but see above, under `'sequenceoffset'`).

`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 timestamp to place in the packet. This enables packets to be placed in the hint track in decoding order, but have their presentation timestamp in the transmitted packet be in a different order. This is necessary for some MPEG payloads.

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.

9.1.3.3 Constructor format

There are various forms of the constructor. Each constructor is 16 bytes, to make iteration easier. The first byte is a union discriminator:

```
aligned(8) class RTPconstructor(type) {
   unsigned int(8)   constructor_type = type;
}

aligned(8) class RTPnoopconstructor
   extends RTPconstructor(0)
{
   uint(8)   pad[15];
}

aligned(8) class RTPimmediateconstructor
   extends RTPconstructor(1)
{
   unsigned int(8)   count;
   unsigned int(8)   data[count];
   unsigned int(8)   pad[14 - count];
}

aligned(8) class RTPsampleconstructor
   extends RTPconstructor(2)
{
   signed int(8)   trackrefindex;
   unsigned int(16)   length;
   unsigned int(32)   samplenumber;
   unsigned int(32)   sampleoffset;
   unsigned int(16)   bytesperblock = 1;
   unsigned int(16)   samplesperblock = 1;
}

aligned(8) class RTPsampledescriptionconstructor
   extends RTPconstructor(3)
{
   signed int(8)   trackrefindex;
   unsigned int(16)   length;
   unsigned int(32)   sampledescriptionindex;
   unsigned int(32)   sampledescriptionoffset;
   unsigned int(32)   reserved;
}
```

The immediate mode permits the insertion of payload-specific headers (e.g. the RTP H.261 header). For hint tracks where the media is sent ‘in the clear’, the `sample` entry then specifies the bytes to copy from the media track, by giving the sample number, data offset, and length to copy. The track reference may index into the table of track references (a strictly positive value), name the hint track itself (-1), or the only associated media track (0). (The value zero is therefore equivalent to the value 1.)

The `bytesperblock` and `samplesperblock` concern compressed audio, using a scheme prior to MP4, in which the audio framing was not evident in the file. These fields have the fixed values of 1 for MP4 files.

The `sampledescription` mode allows sending of sample descriptions (which would contain elementary stream descriptors), by reference, as part of an RTP packet. The index is the index of a `SampleEntry` in a `SampleDescriptionBox`, and the offset is relative to the beginning of that `SampleEntry`.

For complex cases (e.g. encryption or forward error correction), the transformed data would be placed into the hint samples, in the `extradata` field, and then sample mode referencing the hint track itself would be used.

Notice that there is no requirement that successive packets transmit successive bytes from the media stream. For example, to conform with RTP-standard packing of H.261, it is sometimes required that a byte be sent at the end of one packet and also at the beginning of the next (when a macroblock boundary falls within a byte).

9.1.4 SDP information

9.1.4.1 Overview

Streaming servers using RTSP and SDP usually use SDP as the description format; and there are necessary relationships between the SDP information, and the RTP streams, such as the mapping of payload IDs to MIME names. Provision is therefore made for the hinter to leave fragments of SDP information in the file, to assist the server in forming a full SDP description. Note that there are required SDP entries, which the server should also generate. The information here is only partial.

SDP information is formatted as a set of boxes within `UserDataBox`es, at both the movie and the track level. The text in the movie-level SDP box should be placed before any media-specific lines (before the first ‘m=’ in the SDP file).

9.1.4.2 Movie SDP information

At the movie level, within the `UserDataBox`, a hint information container box may occur:

```
aligned(8) class moviehintinformation extends Box('hnti') {
}

aligned(8) class rtpmoviehintinformation extends Box('rtp ') {
   uint(32) descriptionformat = 'sdp ';
   char  sdptext[];
}
```

The `moviehintinformation` box may contain information for multiple protocols; only RTP is defined here. The RTP box may contain information for various description formats; only SDP is defined here. The `sdptext` is correctly formatted as a series of lines, each terminated by <crlf>, as required by SDP.

9.1.4.3 Track SDP information

At the track level, the structure is similar; however, we already know that this track is an RTP hint track, from the sample description. Therefore the child box merely specifies the description format.

```
aligned(8) class trackhintinformation extends Box('hnti') {
}

aligned(8) class rtptracksdphintinformation extends Box('sdp ') {
   char   sdptext[];
}
```

The `sdptext` is correctly formatted as a series of lines, each terminated by <crlf>, as required by SDP.

9.1.5 Statistical information

In addition to the statistics in the hint media header, the hinter may place extra data in a `hintstatisticsbox`, in the track user-data box. This is a container box with a variety of sub-boxes that it may contain.

```
aligned(8) class hintstatisticsbox extends Box('hinf') {
}
aligned(8) class hintBytesSent extends Box('trpy') {
   uint(64)   bytessent; }   // total bytes sent, including 12-byte RTP headers
aligned(8) class hintPacketsSent extends Box('nump') {
   uint(64)   packetssent; }   // total packets sent
aligned(8) class hintBytesSent extends Box('tpyl') {
   uint(64)   bytessent; }   // total bytes sent, not including RTP headers
aligned(8) class hintBytesSent extends Box('totl') {
   uint(32)   bytessent; }   // total bytes sent, including 12-byte RTP headers
aligned(8) class hintPacketsSent extends Box('npck') {
   uint(32)   packetssent; }   // total packets sent
aligned(8) class hintBytesSent extends Box('tpay') {
   uint(32)   bytessent; }   // total bytes sent, not including RTP headers
aligned(8) class hintmaxrate extends Box('maxr') {   // maximum data rate
   uint(32)   period;          // in milliseconds
   uint(32)   bytes; }         // max bytes sent in any period 'period' long
                           //  including RTP headers
aligned(8) class hintmediaBytesSent extends Box('dmed') {
   uint(64)   bytessent; }   // total bytes sent from media tracks
aligned(8) class hintimmediateBytesSent extends Box('dimm') {
   uint(64)   bytessent; }   // total bytes sent immediate mode
aligned(8) class hintrepeatedBytesSent extends Box('drep') {
   uint(64)   bytessent; }   // total bytes in repeated packets
aligned(8) class hintminrelativetime extends Box('tmin') {
   int(32)      time; }      // smallest relative transmission time, milliseconds
aligned(8) class hintmaxrelativetime extends Box('tmax') {
   int(32)      time; }      // largest relative transmission time, milliseconds
aligned(8) class hintlargestpacket extends Box('pmax') {
   uint(32)   bytes; }         // largest packet sent, including RTP header
aligned(8) class hintlongestpacket extends Box('dmax') {
   uint(32)   time; }          // longest packet duration, milliseconds
aligned(8) class hintpayloadID extends Box('payt') {
   uint(32)   payloadID;      // payload ID used in RTP packets
   uint(8)      count;
   char      rtpmap_string[count]; }
```

NOTE It is possible that not all these sub-boxes are present, and that there can be multiple `hintmaxrate` boxes, covering different periods.

9.2 ALC/LCT and FLUTE hint track format

9.2.1 Overview

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 `MetaBox` 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.

FD hint tracks have been designed for the ALC/LCT (asynchronous layered coding/layered coding transport) and FLUTE (file delivery over unidirectional transport) protocols. LCT provides transport level support for reliable content delivery and stream delivery protocols. ALC is a protocol instantiation of the LCT building block, and it serves as a base protocol for massively scalable reliable multicast distribution of arbitrary binary objects. FLUTE builds on top of ALC/LCT and defines a protocol for unidirectional delivery of files.

FLUTE defines a file delivery table (FDT), which carries metadata associated with the files delivered in the ALC/LCT session, and provides mechanisms for in-band delivery and updates of FDT. In contrast, ALC/LCT relies on other means for out-of-band delivery of file metadata, e.g., an electronic service guide that is normally delivered to clients well in advance of the ALC/LCT session combined with update fragments that can be sent during the ALC/LCT session.

File partitionings and FEC reservoirs can be used independently of FD hint tracks and vice versa. The former aid the design of hint tracks and allow alternative hint tracks, e.g., with different FEC overheads, to re-use the same FEC symbols. They also provide means to access source symbols and additional FEC symbols independently for post-delivery repair, which may be performed over ALC/LCT or FLUTE or out-of-band via another protocol. In order to reduce complexity when a server follows hint track instructions, hint tracks refer directly to data ranges of items or data copied into hint samples.

It is recommended that a server sends a different set of FEC symbols for each retransmission of a file.

The syntax for using the `MetaBox` as a container file for source files is defined in 8.10.4, partitions, file and FEC reservoirs are defined in 8.12.8, while the syntax for FD hint tracks is defined in 9.2.

9.2.2 Design principles

The support for file delivery is designed to optimize the server transmission process by enabling ALC/LCT or FLUTE servers to follow simple instructions. It is enough to follow one pre-defined sequence of instructions per channel in order to transmit one session. The file format enables storage of pre-computed source blocks and symbol partitionings, i.e., files may be partitioned into symbols which fit an intended packet size, and pre-computing a certain amount of FEC-symbols that also can be used for post-session repair. The file format also allows storage of alternative ALC/LCT or FLUTE transmission session instructions that may lead to equivalent end results. Such alternatives may be intended for different channel conditions because of higher FEC protection or even by using different error correction schemes. Alternative sessions can refer to a common set of symbols. The hint tracks are flexible and can be used to compose FDT fragments and interleaving of such fragments within the actual object transmission. Several hint tracks can be combined into one or more sessions involving simultaneous transmission over multiple channels.

It is important to make a difference between the definition of sessions for transmission and the scheduling of such sessions. ALC/LCT and FLUTE server files only address optimization of the server transmission process. In order to ensure maximal usage and flexibility of such pre-defined sessions, all details regarding scheduling addresses, etc. are kept outside the definition of the file format. External scheduling applications decide such details, which are not important for optimizing transmission sessions per se. In particular, the following information is out-of-scope of the file format: time scheduling, target addresses and ports, source addresses and ports, and so-called transmission session identifiers (TSI).

The sample numbers associated with the samples of a file delivery hint track provide a numbered sequence. Hint track sample times provide send times of ALC/LCT or FLUTE packets for a default bitrate. Depending on the actual transmission bitrate, an ALC/LCT or FLUTE server may apply linear time scaling. Sample times may simplify the scheduling process, but it is up to the server to send ALC/LCT or FLUTE packets in a timely manner.

A schematic picture of a file containing three alternative hint tracks with different FEC overhead for a source file is provided in Figure 4. In this example, each source block consists of only one sub-block.

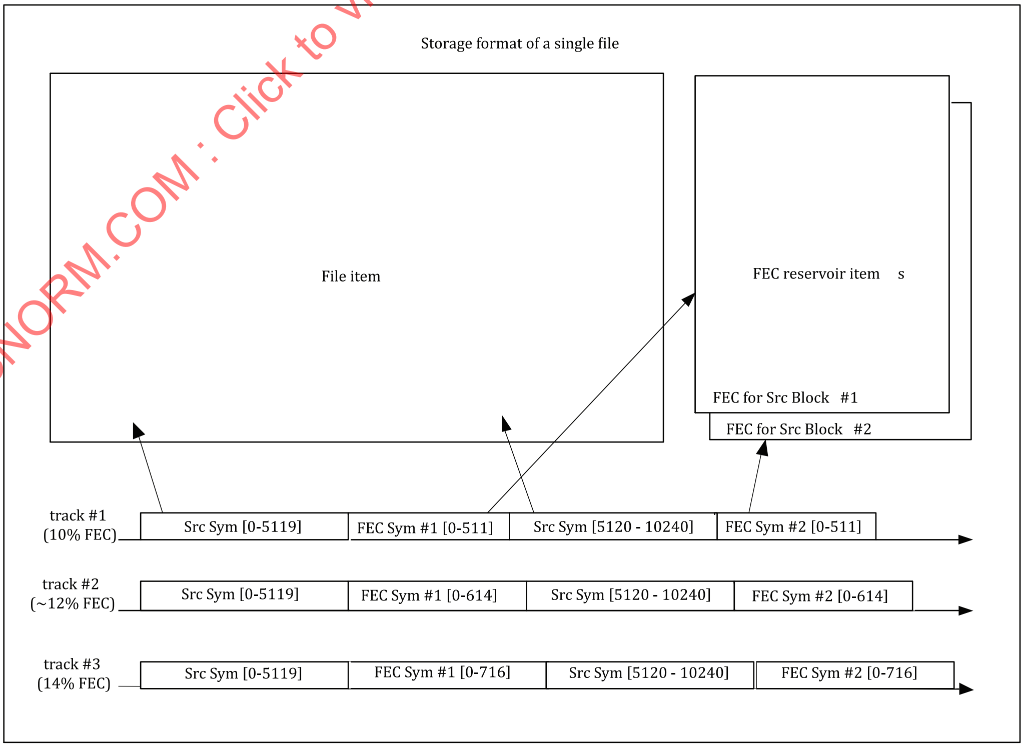

Figure 1 — Different FEC overheads of a source file provided by alternative hint tracks

The source file in Figure 4 is partitioned into 2 source blocks containing symbols of a fixed size. FEC redundancy symbols are calculated for both source blocks and stored as FEC reservoir items. As the hint tracks reference the same items in the file there is no duplication of information. The original source symbols and FEC reservoirs can also be used by repair servers that don't use hint tracks.

9.2.3 Sample description format

9.2.3.1 Definition

FD hint tracks are tracks with `handler_type` `'hint'` and with the entry-format `'fdp '` in the `SampleDescriptionBox`. The FD hint sample entry is contained in the `SampleDescriptionBox`.

9.2.3.2 Syntax

```
class FDHintSampleEntry() extends HintSampleEntry ('fdp ') {
   unsigned int(16)   hinttrackversion = 1;
   unsigned int(16)   highestcompatibleversion = 1;
   unsigned int(16)   partition_entry_ID;
   unsigned int(16)   FEC_overhead;
}
```

9.2.3.3 Semantics

`partition_entry_ID` indicates the partition entry in the `FDItemInformationBox`. A zero value indicates that no partition entry is associated with this sample entry, e.g., for FDT. If the corresponding FD hint track contains only overhead data this value should indicate the partition entry whose overhead data is in question.

`FEC_overhead` is a fixed 8.8 value indicating the percentage protection overhead used by the hint sample(s). The intention of providing this value is to provide characteristics to help a server select a session group (and corresponding FD hint tracks). If the corresponding FD hint track contains only overhead data this value should indicate the protection overhead achieved by using all FD hint tracks in a session group up to the FD hint track in question.

The `hinttrackversion` and `highestcompatibleversion` fields have the same interpretation as in the RTP hint sample entry described in 9.1.2. As additional data a `timescaleentry` box may be provided. If not provided, there is no indication given on timing of packets.

File entries needed for an FDT or an electronic service guide can be created by observing all sample entries of a hint track and the corresponding item information boxes of the items referenced by the above partition entry IDs. No sample entries shall be included in the hint track if they are not referenced by any sample.

9.2.4 Sample format

9.2.4.1 Sample container

Each FD sample in the hint track will generate one or more FD packets.

Each sample contains two areas: the instructions to compose the packets, and any extra data needed when sending those packets (e.g., encoding symbols that are copied into the sample instead of residing in items for source files or FEC). The size of the sample is known from the sample size table.

```
aligned(8) class FDsample extends Box('fdsa') {
   FDPacketBox      packetbox[]
   ExtraDataBox   extradata;      //optional
}
```

Sample numbers of FD samples define the order they shall be processed by the server. Likewise, `FDpacketBox`es in each FD sample should appear in the order they shall be processed. If the `timescaleentry` box is present in the `FDHintSampleEntry`, then sample times are defined and provide relative send times of packets for a default bitrate. Depending on the actual transmission bitrate, a server may apply linear time scaling. Sample times may simplify the scheduling process, but it is up to the server to send packets in a timely manner.

9.2.4.2 Packet entry format

Each packet in the FD sample has the following structure (see IETF RFC 3926,[3] IETF RFC 3450,[4] and IETF RFC 3451[5]):

```
aligned(8) class FDpacketBox extends Box('fdpa') {
   LCTheaderTemplate       LCT_header_info;
   unsigned int(16)       entrycount1;
   LCTheaderExtension    header_extension_constructors[ entrycount1 ];
   unsigned int(16)       entrycount2;
   dataentry             packet_constructors[ entrycount2 ];
}
```

The LCT header info contains LCT header templates for the current FD packet. Header extension constructors are structures which are used for constructing the LCT header extensions. Packet constructors are used for constructing the FEC payload ID and the source symbols in an FD packet.

9.2.4.3 LCT header template format

The LCT header template is defined as follows:

```
aligned(8) class LCTheaderTemplate {
   unsigned int(1)    sender_current_time_present;
   unsigned int(1)    expected_residual_time_present;
   unsigned int(1)    session_close_bit;
   unsigned int(1)    object_close_bit;
   unsigned int(4)    reserved;
   unsigned int(16)    transport_object_identifier;
}
```

It can be used by a server to form an LCT header for a packet. Note that some parts of the header depend on the server policy and are not included in the template. Some field lengths also depend on the LCT header bits assigned by the server. The server may also need to change the value of the transport object identifier (TOI).

9.2.4.4 LCT header extension constructor format

The LCT header extension constructor format is defined as follows:

```
aligned(8) class LCTheaderextension {
   unsigned int(8) header_extension_type;
   if (header_extension_type > 127) {
      unsigned int(8) content[3];
   }
   else {
   unsigned int(8) length;
   if (length > 0) {
      unsigned int(8) content[(length*4) - 2];
   }
}
```

A positive value of the length field specifies the length of the constructor content in multiples of 32 bit words. A zero value means that the header is generated by the server.

The usage and rules for LCT header extensions are defined in IETF RFC 3451[5] (LCT RFC). The `header_extension_type` contains the LCT header extension type (HET) value.

HET values between 0 and 127 are used for variable-length (multiple 32-bit word) extensions. HET values between 128 and 255 are used for fixed length (one 32-bit word) extensions. If the `header_extension_type` is smaller than 128, then the length field corresponds to the LCT header extension length (HEL) as defined in IETF RFC 3451[5]. The content field always corresponds to the header extension content (HEC).

NOTE A server can identify packets including FDT by observing whether EXT_FDT (`header_extension_type == 192`) is present.

9.2.4.5 Packet constructor format

There are various forms of the constructor. Each constructor is 16 bytes in order to make iteration easier. The first byte is a union discriminator. The packet constructors are used to include FEC payload ID as well as source and parity symbols in an FD packet.

```
aligned(8) class FDconstructor(type) {
   unsigned int(8)   constructor_type = type;
}

aligned(8) class FDnoopconstructor extends FDconstructor(0)
{
   unsigned int(8)   pad[15];
}

aligned(8) class FDimmediateconstructor extends FDconstructor(1)
{
   unsigned int(8)   count;
   unsigned int(8)   data[count];
   unsigned int(8)   pad[14 - count];
}

aligned(8) class FDsampleconstructor extends FDconstructor(2)
{
   signed int(8)      trackrefindex;
   unsigned int(16)   length;
   unsigned int(32)   samplenumber;
   unsigned int(32)   sampleoffset;
   unsigned int(16)   bytesperblock = 1;
   unsigned int(16)   samplesperblock = 1;
}

aligned(8) class FDitemconstructor extends FDconstructor(3)
{
   unsigned int(16)   item_ID;
   unsigned int(16)   extent_index;
   unsigned int(64)   data_offset;   //offset in byte within extent
   unsigned int(24)   data_length;   //non-zero length in bytes within extent or
                                    //if (data_length==0) rest of extent
}
aligned(8) class FDitemconstructorLarge extends FDconstructor(5)
{
   unsigned int(32)   item_ID;
   unsigned int(32)   extent_index;
   unsigned int(64)   data_offset;   //offset in byte within extent
   unsigned int(24)   data_length;   //non-zero length in bytes within extent or
                                    //if (data_length==0) rest of extent
}
aligned(8) class FDxmlboxconstructor extends FDconstructor(4)
{
   unsigned int(64)   data_offset; //offset in byte within XMLBox or BinaryXMLBox
   unsigned int(32)   data_length;
   unsigned int(24)   reserved;
}
```

9.2.4.6 Extra data box

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

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

9.2.4.7 FEC information box

9.2.4.7.1 Definition

Box Type: 'feci'
Container: ExtraDataBox
Mandatory: No
Quantity: Zero or One

The FECInformationBox 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 IETF RFC 5052), in which (i) value zero corresponds to the "Compact No-Code FEC scheme" also known as "Null-FEC" (IETF RFC 3695[6]); (ii) value one corresponds to the “MBMS FEC” (3GPP TS 26.346[2]); (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 IETF 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.

9.3 MPEG-2 transport hint track format

9.3.1 Overview

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

9.3.2.1 General 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.

9.3.2.2 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.

9.3.2.3 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 `TSTimingBox` 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.

9.3.2.4 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 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.

9.3.2.5 Random-access points

A sync sample 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 that is 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 1 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 2 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.

9.3.2.6 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 can 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.

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 As with server hint tracks, the sample formats of reception hint tracks can enable construction of packets by pulling data out of other tracks by reference. These other tracks could 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 can 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 will need to 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).

A sample with a size of zero is permitted in reception hint tracks, and such samples may be ignored.

9.3.3 Sample description format

9.3.3.1 Definition

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;
}
```

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 PATBox or the PMTBox 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 PATBox contains the section data of the PAT and each PMTBox contains the section data of one of the PMTs.

In the case of an SPTS, it is strongly recommended that the `PMTBox` 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 `PMTBox` 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 `PATBox` 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

9.3.4.1 Definition

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.

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 1 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 can be variable or restricted, e.g. extensions of this file format can define a sample to contain exactly one packet.

NOTE 2 It is possible to compact common sequences of bytes in transport packets by including those bytes in one or more packets directly for example in their `precedingbytes` or `trailingbytes` section, and then using the MPEG2TSSampleConstructor in other places to refer to them; this is especially relevant for runs of 0xFF bytes.

9.3.4.2 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 || sync_byte == 0x01) {
      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 (sync_byte == 0x00 && (adaptation_field_control == ´10´ ||
         adaptation_field_control == ´11´)) {
         uint(8)      adaptation_field[headerdatalen-3];
      }
      MPEG2TSConstructor   constructors[num_constructors];
   } else if (sync_byte == 0xFF) {
      // implicit null packet that has been removed
   }
   uint(8)   trailingbytes[trailingbyteslen];
}
// Sample format
aligned(8) class MPEG2TSSample {
   MPEG2TSPacketRepresentation   sample[];
}
```

9.3.4.3 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 packet representation contains a transport stream packet (a precomputed reception hint track sample), with the remaining bytes following in the field `packet`. The values 0x00 and 0x01 are used for constructed packet representation(s). If MPEG2TSSampleConstructor is used to construct packet representation(s), it points to a track indexed by `trackrefindex` in the `TrackReferenceBox` with reference type `'hint'`. If this value is 0xFF, it implies that a null packet has been removed at this position. All other values are currently reserved.

`trackrefindex` indexes in the `TrackReferenceBox` 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` shall 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[15].

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

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 in ISO/IEC 13818-1[15].

9.3.5 Protected MPEG 2 transport stream hint track

9.3.5.1 Overview

This subclause 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 MPEG2TSSampleEntry('pm2t') {
   ProtectionSchemeInfoBox      SchemeInformation;
}
```

9.3.5.3 Semantics

The `ProtectionSchemeInfoBox` (defined in 8.12.2) shall contain details of the protection scheme applied. This shall include the `OriginalFormatBox` which shall contain the original sample entry type of the `MPEG2TSSampleEntry`.

9.4 RTP, RTCP, SRTP and SRTCP reception hint tracks

9.4.1 RTP reception hint track

9.4.1.1 Overview

This subclause specifies the reception hint track format for the real-time transport protocol (RTP), as defined in IETF RFC 3550[9].

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 HintSampleEntry ('rrtp') {
   uint(16)      hinttrackversion = 1;
   uint(16)      highestcompatibleversion = 1;
   uint(32)      maxpacketsize;
}
```

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 timescaleentry and timeoffset boxes. Moreover, the additionaldata may contain a timestamp synchrony box.

The timescaleentry box 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 timeoffset may be present. If the timeoffset 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 timeoffset box should therefore be present in RTP reception hint tracks and the value of offset in the timeoffset 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 Formula 1 (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 Formula 1 (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 Formula 1 (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 shall 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. 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.

A sample with a size of zero is permitted in reception hint tracks, and such samples may be ignored.

9.4.1.4 Packet entry format

Each packet in the packet entry table has same structure as for server (transmission) hint tracks, in 9.1.3.2.

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 TimeToSampleBox 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 TimeToSampleBox. In this case timestamp_sync is set to 1, the sample durations are mostly constant, and the timeoffset 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 shall 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 9.1.3.2.

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 timeoffset box of the referred reception hint sample entry, and % be the modulo operation. The value of offset shall be such that Formula (1) is true:

$$RTPtimestamp = (DT_i + tsro.offset + offset) \bmod 2^{32} \quad (1)$$

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 TimeToSampleBox 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 might 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 EditListBoxes, 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 nth entry of `CSRC[]` shall equal the nth 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 Overview

This subclause specifies the reception hint track format for the real-time control protocol (RTCP), defined in IETF RFC 3550[9].

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 `timestampsynchrony` 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 `TrackReferenceBox` 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 `MediaHeaderBox` of an RTCP reception hint track shall be equal to the value of `timescale` in the `MediaHeaderBox` 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 Overview

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. 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 IETF RFC 3550[9]), 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. 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 Overview

When reception hint tracks are used to store secure real-time transport protocol (SRTP) streams, as defined in IETF RFC 3711, the formats in this subclause shall be used.

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 HintSampleEntry ('rsrp') {
   uint(16)      hinttrackversion = 1;
   uint(16)      highestcompatibleversion = 1;
   uint(32)      maxpacketsize;
}
```

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 `ReceivedSsrcBox`.

Additionally, the additionaldata[] may contain the ReceivedCryptoContextIdBox and the RolloverCounterBox defined below. Furthermore, an SRTPProcessBox 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 RolloverCounterBox as IETF RFC 4771[11] 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.

9.4.4 SRTCP reception hint tracks

9.4.4.1 Overview

When reception hint tracks are used to store secure real-time control protocol (SRTCP) streams, as defined in IETF RFC 3711, the formats in this subclause shall be used.

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, in comparison to 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 `timestampsynchrony` 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 `TrackReferenceBox` 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 `MediaHeaderBox` of an SRTCP reception hint track shall be equal to the value of `timescale` in the `MediaHeaderBox` 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 `SRTPProcessBox` 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 Overview

This document 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 `ProtectionSchemeInfoBox` 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 `ReceivedRtpHintSampleEntry`).

9.4.6 Recording procedure

See Annex H.

9.4.7 Parsing procedure

See Annex H.

10 Sample groups

10.1 Random access recovery points

10.1.1 Definition

In some coding systems it is possible to random access into a stream and achieve correct decoding after having decoded a number of samples. This is known as gradual decoding refresh. For example, in video, the encoder might encode intra-coded macroblocks in the stream, such that it knows that within a certain period the entire picture consists of pixels that are only dependent on intra-coded macroblocks supplied during that period.

Samples for which such gradual refresh is possible are marked by being a member of one of these groups. The definition of the groups allows the marking to occur at either the beginning of the period or the end. However, when used with a particular media type, the usage of these groups may be restricted to marking only one end (i.e. restricted to only positive or negative roll values). A roll-group is defined as that group of samples having the same roll distance.

The roll groups have the following semantics.

A `VisualRollRecoveryEntry` documents samples that enable entry points into streams that are alternatives to sync samples.

An `AudioRollRecoveryEntry` documents the pre-roll distance required in audio streams in which every sample can be independently decoded, but the decoder output is only assured to be correct after pre-rolling by the indicated number of samples.

An `AudioPreRollEntry` documents samples that enable entry points into streams that are independently decodable and are thus alternatives to sync samples. It should be used with audio streams in which not every sample can be independently decoded; decoding can only start at an independently decodable sample and decoder output is only assured to be correct after pre-rolling by the indicated number of samples.

The `roll_distance` shall be a positive value. Decoding starts at a member sample, but decoder output is only assured to be correct after decoding the indicated number of samples.

10.1.2 Syntax

```
class VisualRollRecoveryEntry() extends VisualSampleGroupEntry ('roll')
{
   signed int(16) roll_distance;
}
class AudioRollRecoveryEntry() extends AudioSampleGroupEntry ('roll')
{
   signed int(16) roll_distance;
}
class AudioPreRollEntry() extends AudioSampleGroupEntry ('prol')
{
   signed int(16) roll_distance;
}
```

10.1.3 Semantics

`roll_distance` is a signed integer that gives the number of samples that need to be decoded in order for a sample to be decoded correctly. A positive value indicates the number of samples after the sample that is a group member that need to be decoded such that at the last of these recovery is complete, i.e. the last sample is correct. A negative value indicates the number of samples before the sample that is a group member that need to be decoded in order for recovery to be complete at the marked sample. The value zero shall not be used; those samples that would be covered by the value zero can be signalled by other signaling mechanisms such as the sync sample table, the `'rap '` sample grouping and the `'sap '` sample grouping.

10.2 Rate share groups

10.2.1 Overview

Rate share instructions are used by players and streaming servers to help allocating bitrates dynamically when several streams share a common bandwidth resource. The instructions are stored in the file as sample group entries and apply when scalable or alternative media streams at different bitrates are combined with other scalable or alternative tracks. The instructions are time-dependent as samples in a track may be associated with different sample group entries. In the simplest case, only one target rate share value is specified per media and time range as illustrated in Figure 2.

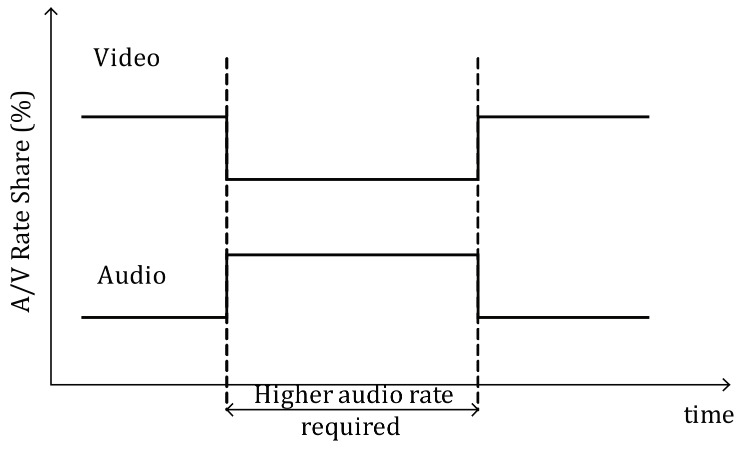

Figure 2 — Audio/Video rate share as function of time

In order to accommodate for rate share values that vary with the available bitrate, it is possible to specify more than one operation range. One may for instance indicate that audio requires a higher percentage (than video) at low available bitrates. Technically this is done by specifying two operation points as shown in Figure 3.

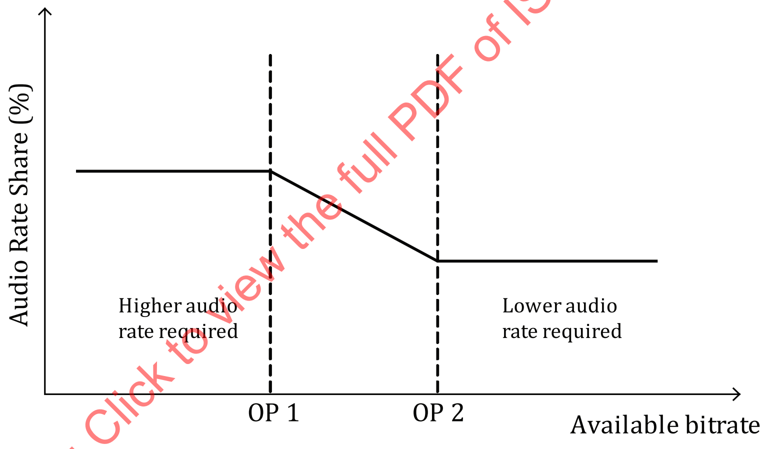

Figure 3 — Audio rate share as function of available bitrate

Operation points are defined in terms of total available bandwidth. For more complex situations it is possible to specify more operation points.

In addition to target rate share values, it is also possible to specify maximum and minimum bitrates for a certain media, as well as discard priority.

10.2.2 Rate share sample group entry

10.2.2.1 Definition

Each sample of a track may be associated to (zero or) one of a number of sample group descriptions, each of which defines a record of rate-share information. Typically the same rate-share information applies to many consecutive samples and it may therefore be enough to define two or three sample group descriptions that can be used at different time intervals.

The `grouping_type` `'rash'` (short for rate share) is defined as the grouping criterion for rate share information. Zero or one `SampleToGroupBox`es for the `grouping_type` `'rash'` can be contained in the `SampleTableBox` of a track. It shall reside in a hint track, if a hint track is used, otherwise in a media track.

Target rate share may be specified for several operation points that are defined in terms of the total available bitrate, i.e., the bitrate that should be shared. If only one operation point is defined, the target rate share applies to all available bitrates. If several operation points are defined, then each operation point specifies a target rate share. Target rate share values specified for the first and the last operation points also specify the target rate share values at lower and higher available bitrates, respectively. The target rate share between two operation points is specified to be in the range between the target rate shares of those operation points. One possibility is to estimate with linear interpolation.

10.2.2.2 Syntax

```
class RateShareEntry() extends SampleGroupDescriptionEntry('rash') {
   unsigned int(16)   operation_point_count;
   if (operation_point_count == 1) {
      unsigned int(16)      target_rate_share;
   }
   else {
      for (i=0; i < operation_point_count; i++) {
         unsigned int(32)   available_bitrate;
         unsigned int(16)   target_rate_share;
      }
   }
   unsigned int(32)   maximum_bitrate;
   unsigned int(32)   minimum_bitrate;
   unsigned int(8)   discard_priority;
}
```

10.2.2.3 Semantics

`operation_point_count` is a non-zero integer that gives the number of operation points.

`available_bitrate` is a positive integer that defines an operation point (in kilobits per second). It is the total available bitrate that can be allocated in shares to tracks. Each entry shall be greater than the previous entry.

`target_rate_share` is an integer. A non-zero value indicates the percentage of available bandwidth that should be allocated to the media for each operation point. The value of the first (last) operation point applies to lower (higher) available bitrates than the operation point itself. The target rate share between operation points is bounded by the target rate shares of the corresponding operation points. A zero value indicates that no information on the preferred rate share percentage is provided.

`maximum_bitrate` is an integer. A nonzero value indicates (in kilobits per second) an upper threshold for which bandwidth should be allocated to the media. A higher bitrate than maximum bitrate should only be allocated if all other media in the session has fulfilled their quotas for target rate-share and maximum bitrate, respectively. A zero value indicates that no information on maximum bitrate is provided.

`minimum_bitrate` is an integer. A nonzero value indicates (in kilobits per second) a lower threshold for which bandwidth should be allocated to the media. If the allocated bandwidth would correspond to a smaller value, then no bitrate should be allocated. Instead preference should be given to other media in the session or alternate encodings of the same media. Zero minimum bitrate indicates that no information on minimum bitrate is provided.

`discard_priority` is an integer indicating the priority of the track when tracks are discarded to meet the constraints set by target rate share, maximum bitrate and minimum bitrate. Tracks are discarded in discard priority order and the track that has the highest discard priority value is discarded first.

10.2.3 Relationship between tracks

The purpose of defining rate share information is to aid a server or player extracting data from a track in combination with other tracks. Note that a server/player streams/plays tracks simultaneously if they belong to different alternate groups and can switch between tracks that belong to the same switch group within an alternate group. By default, all tracks are served/played simultaneously if no alternate groups are defined.

Rate share information should be provided for each track. A track that does not include rate share information has one operation point and can be treated as a constant-bitrate track with discard priority 128. Target rate share, minimum and maximum bitrates do not apply in this case.

Tracks that are alternates to each other shall (at each instance of time) define the same number of operation points at the same set of total available bitrates and have the same discard priorities. Note that the number and definition of operation points may depend on time. Alternate tracks may have different target rate shares, minimum and maximum bitrates.

10.2.4 Bitrate allocation

Rate share information on maximum bitrate, minimum bitrate, and target rate share can be combined for a track. If this is the case, the target rate share shall be applied to find an allocated bitrate before the impact of the maximum and minimum bitrates is considered.

When allocating bandwidth to several tracks, the following considerations apply:

1. In the case all tracks have explicit target rate share values and they don't sum up to 100 per cent, treat them as weights, i.e., normalize them.
2. The total allocation shall not exceed total available bitrate.
3. In a choice between alternate tracks, the chosen track should be the track that causes the alternate group to have an allocation most closely in accord with its target rate share, or the track that desires the highest bitrate that can be allocated without discarding other tracks (see below).
4. Tracks must have an allocation between their minimum and maximum bitrates, or be discarded.
5. Tracks should have an allocation in accord with their target rate shares, but this may be distorted to allow some tracks to achieve their minima, or in case some have reached their maxima.
6. If an allocation cannot be done including a track from every alternate group, then tracks should be discarded in discard priority order.
7. The allocation must be re-calculated whenever the operating set for an active track (one that has been selected from an alternate group) changes or the available bitrate changes.

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 or a sample marked by `'rap '` sample grouping, which are collectively referred to as the initial sample below. 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 `SampleToGroupBox` may be used with the alternative startup sequence sample grouping. If version 1 of the `SampleToGroupBox` is used, 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, an initial sync sample from which to start decoding is identified by using the `SyncSampleBox`, the `sample_is_non_sync_sample` flag for samples enclosed in track fragments, or the `'rap '` sample grouping. Then, if the initial 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 of the sync initial 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 `TimeToSampleBox` or the `TrackFragmentHeaderBox`. The sync initial 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 composition order. 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 4 illustrates a typical hierarchically scalable bitstream with five temporal levels. Figure 4a shows the example sequence in composition 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 4b shows the example sequence in decoding order. Figure 4c shows the example sequence in composition order when assuming that the composition 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.

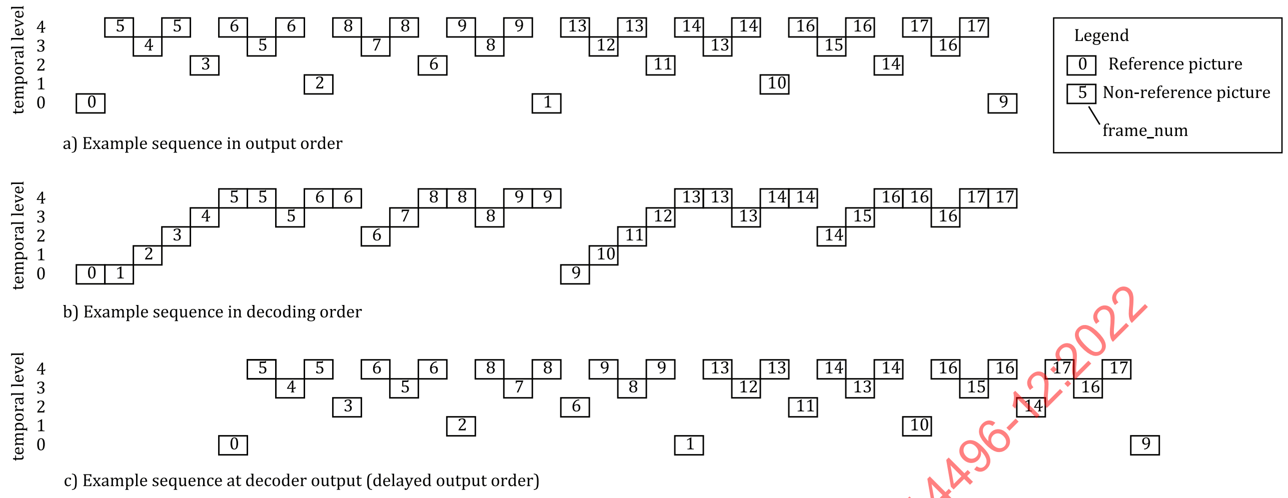
temporal level
a) Example sequence in output order
b) Example sequence in decoding order
c) Example sequence at decoder output (delayed output order)
Legend
Reference picture
Non-reference picture
frame_num

Figure 4 — 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 5 and Figure 6 show two examples of alternative startup sequences where a subset of the bitstream of Figure 4 is decoded.

The samples selected for decoding and the decoder output are presented in Figure 5a and Figure 5b, 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 4. 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.

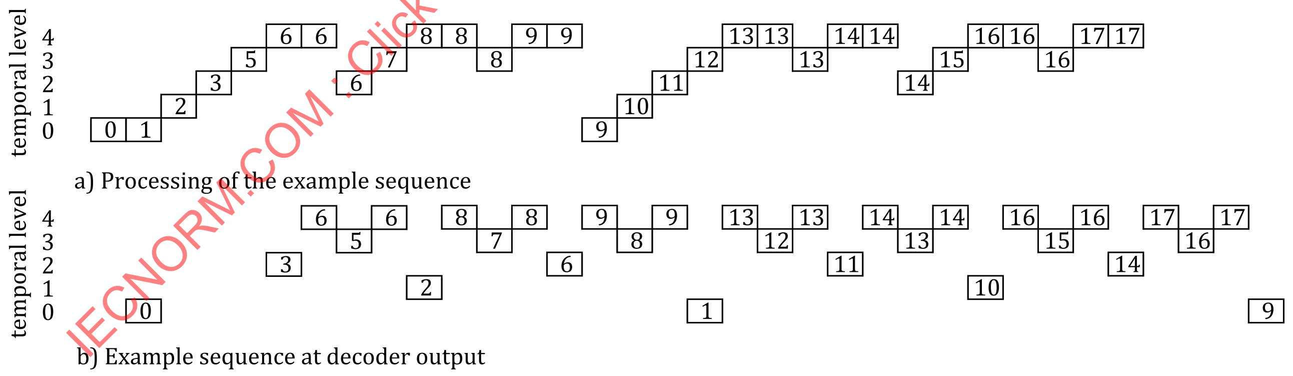
temporal level
a) Processing of the example sequence
b) Example sequence at decoder output

Figure 5 — An example of an alternative startup sequence

In the example of Figure 6, 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 4.

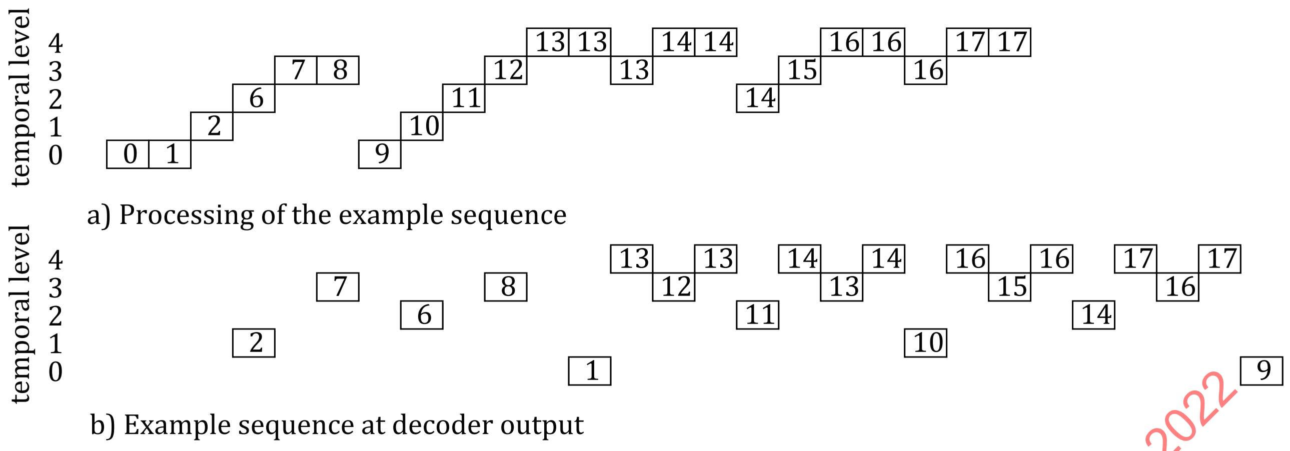
temporal level
4
3
2
1
0
0
1
2
6
7
8
9
10
11
12
13
13
13
14
14
14
15
16
16
16
17
17
a) Processing of the example sequence
temporal level
b) Example sequence at decoder output

Figure 6 — Another example of an alternative startup sequence

10.4 Random access point (RAP) sample group

10.4.1 Definition

“Open” random-access samples can be marked by being a member of this group. Samples marked by this group shall be random access points, and may also be sync points (i.e. it is not required that samples marked by the sync sample table be excluded).

10.4.2 Syntax

```
class VisualRandomAccessEntry() extends VisualSampleGroupEntry ('rap ')
{
   unsigned int(1) num_leading_samples_known;
   unsigned int(7) num_leading_samples;
}
```

10.4.3 Semantics

`num_leading_samples_known` equal to 1 indicates that the number of leading samples is known for each sample in this group, and the number is specified by num_leading_samples.

`num_leading_samples` specifies the number of leading samples for each sample in this group. When num_leading_samples_known is equal to 0, this field should be ignored.

10.5 Temporal level sample group

10.5.1 Definition

Many video codecs support temporal scalability where it is possible to extract one or more subsets of frames that can be independently decoded. A simple case is the extraction of I frames for a bitstream with a regular I-frame interval, e.g,, IPPPIPPP..., where every 4th picture is an I frame. Also subsets of these I frames can be extracted for even lower frame rates. More elaborate situations with several temporal levels can be constructed using hierarchical B or P frames.

The temporal level sample grouping (`'tele'`) provides a codec-independent sample grouping that can be used to group samples (access units) in a track (and potential track fragments) according to temporal level, where samples of one temporal level have no coding dependencies on samples of higher temporal levels. The temporal level equals the sample group description index (taking values 1, 2, 3, etc). The bitstream containing only the access units from the first temporal level to a higher temporal level remains conforming to the coding standard.

A grouping according to temporal level facilitates easy extraction of temporal subsequences, for instance using the SubsegmentIndexBox in 8.16.4.

10.5.2 Syntax

```
class TemporalLevelEntry() extends VisualSampleGroupEntry('tele')
{
   bit(1)    level_independently_decodable;
   bit(7)    reserved=0;
}
```

10.5.3 Semantics

The temporal level of samples in a sample group equals to the sample group description index.

level_independently_decodable is a flag. 1 indicates that all samples of this level have no coding dependencies on samples of other levels. 0 indicates that no information is provided.

10.6 Stream access point sample group

10.6.1 Definition

A stream access point, as defined in Annex I, enables random access into a container of media stream(s). The SAP sample grouping identifies samples (the first byte of which is the position I_{SAU} for a SAP as specified in Annex I) as being of the indicated SAP type.

The syntax and semantics of grouping_type_parameter are specified as follows.

```
{
   unsigned int(28)   target_layers;
   unsigned int(4)   layer_id_method_idc;
}
```

target_layers specifies the target layers for the indicated SAPs according to Annex I. The semantics of target_layers depends on the value of layer_id_method_idc. When layer_id_method_idc is equal to 0, target_layers is reserved.

layer_id_method_idc specifies the semantics of target_layers. layer_id_method_idc equal to 0 specifies that the target layers consist of all the layers represented by the track. layer_id_method_idc not equal to 0 is specified by derived media format specifications.

10.6.2 Syntax

```
class SAPEntry() extends  SampleGroupDescriptionEntry('sap ')
{
   unsigned int(1) dependent_flag;
   unsigned int(3) reserved;
   unsigned int(4) SAP_type;
}
```

10.6.3 Semantics

reserved shall be equal to 0. Parsers shall allow and ignore all values of reserved.

dependent_flag shall be 0 for non-layered media. dependent_flag equal to 1 specifies that the reference layers, if any, for predicting the target layers may have to be decoded for accessing a sample of this sample group. dependent_flag equal to 0 specifies that the reference layers, if any, for predicting the target layers need not be decoded for accessing any SAP of this sample group.

sap_type values equal to 0 and 7 are reserved; sap_type values in the range of 1 to 6, inclusive, specify the SAP type, as specified in Annex I, of the associated samples (for which the first byte of a sample in this group is the position I_{SAU}).

10.7 Sample-to-item sample group

10.7.1 Definition

Samples of a track can be linked to one more metadata items using the sample-to-item sample grouping. The `MetaBox` containing the referred items is resolved as specified in the semantics below.

The sample-to-item sample grouping is allowed for any types of tracks, and its syntax and semantics are unchanged regardless of the track handler type.

In the absence of this sample group, the entire track-level `MetaBox`, if any, is applicable to every sample.

10.7.2 Syntax

```
class SampleToMetadataItemEntry()
extends SampleGroupDescriptionEntry('stmi') {
   unsigned int(32) meta_box_handler_type;
   unsigned int(32) num_items;
   for(i = 0; i < num_items; i++) {
      unsigned int(32) item_id[i];
   }
}
```

10.7.3 Semantics

`meta_box_handler_type` informs about the type of metadata schema used by the `MetaBox` which is referenced by the items in this sample group. When there are multiple `MetaBoxes` with the same handler types, the `MetaBox` referred to in this sample group entry is the first `MetaBox` fulfilling one of the following ordered constraints:

— A `MetaBox` included in the current track, with `handler_type` equal to `meta_box_handler_type`.

— A `MetaBox` included in `MovieBox`, with `handler_type` equal to `meta_box_handler_type`.

— A `MetaBox` included in the root level of the file, with `handler_type` equal to `meta_box_handler_type`.

`num_items` counts the number of items referenced by this sample group.

`item_id[i]` specifies the `item_ID` value of an item that applies to or is valid for the sample mapped to this sample group description entry.

10.8 Dependent random access point (DRAP) sample group

10.8.1 Definition

A dependent random access point (DRAP) sample is a sample after which all samples in decoding order can be correctly decoded if the closest *initial* sample preceding the DRAP sample is available for reference. The initial sample is a SAP sample of SAP type 1, 2 or 3 that is marked as such either by being a Sync sample or by the SAP sample group. For example, if the 32nd sample in a file is an initial sample consisting of an I-picture, the 48th sample may consist of a P-picture and be marked as a member of the dependent random access point sample group, thereby indicating that random access can be performed at the 48th sample by first decoding the 32nd sample (ignoring samples 33-47) and then continuing to decode from the 48th sample.

A sample can be a member of the dependent random access point Sample Group (and hence called a DRAP sample) only if the following conditions are true

— The DRAP sample references only the closest preceding initial sample.

— The DRAP sample and all samples following the DRAP sample in output order can be correctly decoded when starting decoding at the DRAP sample after having decoded the closest preceding SAP sample of type 1, 2 or 3 marked as such by being a Sync sample or by the SAP sample group.

NOTE DRAP samples can only be used in combination with SAP samples of type 1, 2 and 3. This is in order to enable the functionality of creating a decodable sequence of samples by concatenating the preceding SAP sample with the DRAP sample and the samples following the DRAP sample in output order

10.8.2 Syntax

```
class VisualDRAPEntry()
extends VisualSampleGroupEntry('drap') {
   unsigned int(3) DRAP_type;
   unsigned int(29) reserved = 0;
}
```

10.8.3 Semantics

`DRAP_type` is a non-negative integer. When `DRAP_type` is in the range of 1 to 3 it indicates the `SAP_type` (as specified in Annex I) that the DRAP sample would have corresponded to, had it not depended on the closest preceding SAP. Other type values are reserved.

`reserved` shall be equal to 0. The semantics of this subclause only apply to sample group description entries with `reserved` equal to 0. Parsers shall allow and ignore sample group description entries with `reserved` greater than 0 when parsing this sample group.

10.9 Pixel Aspect Ratio Sample Grouping

10.9.1 Definition

The Pixel Aspect Ratio sample group (`'pasr'`) may be used to signal the pixel aspect ratio of samples in a video track, when the pixel aspect ratio of the samples within a track change dynamically and a single value in a `PixelAspectRatioBox` in a sample entry, specified in 12.1.4 cannot therefore be used.

When the Pixel Aspect Ratio sample group is used in a track, the `PixelAspectRatioBox` shall not be present in any sample entry of that track.

10.9.2 Syntax

```
class PixelAspectRatioEntry() extends VisualSampleGroupEntry ('pasr'){
   unsigned int(32) hSpacing;
   unsigned int(32) vSpacing;
}
```

10.9.3 Semantics

`hSpacing`, `vSpacing`: define the relative width and height of a pixel as defined for the `PixelAspectRatioBox` in 12.1.4

10.10 Clean Aperture Sample Grouping

10.10.1 Definition

The Clean Aperture sample group (`'casg'`) may be used to signal the clean aperture of samples in a video track, when the clean aperture of the samples within a track change dynamically and a single value in a CleanApertureBox in a sample entry, specified in 12.1.4 cannot therefore be used.

When the Clean Aperture sample group is used in a track, the `CleanApertureBox` shall not be present in any sample entry of that track.

10.10.2 Syntax

```
class CleanAperture Entry() extends VisualSampleGroupEntry ('casg'){
   unsigned int(32) cleanApertureWidthN;
   unsigned int(32) cleanApertureWidthD;

   unsigned int(32) cleanApertureHeightN;
   unsigned int(32) cleanApertureHeightD;


   unsigned int(32) horizOffN;
   unsigned int(32) horizOffD;


   unsigned int(32) vertOffN;
   unsigned int(32) vertOffD;

}
```

10.10.3 Semantics

`cleanApertureWidthN`, `cleanApertureWidthD`, `cleanApertureHeightN`, `cleanApertureHeightD`, `horizOffN`, `horizOffD`, `vertOffN` and `vertOffD` define the clean aperture width, height and horizontal and vertical offsets of the clean aperture center as defined for the `CleanApertureBox` in 12.1.4

11 Derived file formats

This document may be used as the basis of a specific file format for a restricted purpose: for example, the MP4 file format for MPEG-4 and the Motion JPEG 2000 file format are both derived from it. When a derived specification is written, the following must be specified:

— The name of the new format, and its brand and compatibility types for the `FileTypeBox`. Generally a new file extension will be used, a new MIME type, and Macintosh file type also, though the definition and registration of these are outside the scope of this document.

— Any template fields used must be explicitly declared; their use must be conformant with this document.

— The exact `codingname` and `protocol` identifiers as used in a `SampleEntry` must be defined. The format of the samples that these code points identify must also be defined. However, it may be preferable to fit the new coding systems into an existing framework (e.g. the MPEG-4 systems framework), than to define new coding points at this level. For example, a new audio format could use a new `codingname`, or could use `'mp4a'` and register new identifiers within the MPEG-4 audio framework.

New boxes may be defined, though this is discouraged.

If the derived specification needs a new track type other than those defined here or registered, then a new handler-type must be registered. The media header required for this track must be identified. If it is a new box, it must be defined and its box type registered. In general, it is expected that most systems can use existing track types.

Any new track reference types should be registered and defined.

As defined above, the Sample Description format may be extended with optional or required boxes. The usual syntax for doing this would be to define a new box with a specific name, extending (for example) Visual Sample Entry, and containing new boxes.

12 Media-specific definitions

12.1 Video media

12.1.1 Media handler

Video media uses the 'vide' handler type in the HandlerBox of the MediaBox, as defined in 8.4.3.

Auxiliary video media uses the 'auxv' handler type in the HandlerBox of the MediaBox, as defined in 8.4.3.

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.

12.1.2 Video media header

12.1.2.1 Definition

Box Types: 'vmhd'
Container: MediaInformationBox
Mandatory: Yes
Quantity: Exactly one

Video tracks use the VideoMediaHeaderBox in the MediaInformationBox as defined in 8.4.5. The VideoMediaHeaderBox contains general presentation information, independent of the coding, for video media. Note that the flags field has the value 1.

12.1.2.2 Syntax

```
aligned(8) class VideoMediaHeaderBox
   extends FullBox('vmhd', version = 0, 1) {
   template unsigned int(16)      graphicsmode = 0;    // copy, see below
   template unsigned int(16)[3]   opcolor = {0, 0, 0};
}
```

12.1.2.3 Semantics

version is an integer that specifies the version of this box

graphicsmode specifies a composition mode for this video track, from the following enumerated set, which may be extended by derived specifications:

copy = 0 copy over the existing image

opcolor is a set of 3 colour values (red, green, blue) available for use by graphics modes

12.1.3 Sample entry

12.1.3.1 Definition

Video tracks use VisualSampleEntry.

In video tracks, the frame_count field shall be 1 unless the specification for the media format explicitly documents this template field and permits larger values. That specification must document both how the individual frames of video are found (their size information) and their timing established. That timing might be as simple as dividing the sample duration by the frame count to establish the frame duration.

The width and height in the video sample entry document the pixel counts that the codec will deliver; this enables the allocation of buffers. Since these are counts they do not take into account pixel aspect ratio.

12.1.3.2 Syntax

```
class VisualSampleEntry(codingname) extends SampleEntry (codingname){
   unsigned int(16) pre_defined = 0;
   const unsigned int(16) reserved = 0;
   unsigned int(32)[3]   pre_defined = 0;
   unsigned int(16)   width;
   unsigned int(16)   height;
   template unsigned int(32)   horizresolution = 0x00480000;   // 72 dpi
   template unsigned int(32)   vertresolution  = 0x00480000;   // 72 dpi
   const unsigned int(32)   reserved = 0;
   template unsigned int(16)   frame_count = 1;
   uint(8)[32]   compressorname;
   template unsigned int(16)   depth = 0x0018;
   int(16)   pre_defined = -1;
   // other boxes from derived specifications
   CleanApertureBox         clap;      // optional
   PixelAspectRatioBox      pasp;      // optional
}
```

12.1.3.3 Semantics

`resolution` fields give the resolution of the image in pixels-per-inch, as a fixed 16.16 number

`frame_count` indicates how many frames of compressed video are stored in each sample. The default is 1, for one frame per sample; it may be more than 1 for multiple frames per sample

`compressorname` is a name, for informative purposes. It is formatted in a fixed 32-byte field, with the first byte set to the number of bytes to be displayed, followed by that number of bytes of displayable data encoded using UTF-8, and then padding to complete 32 bytes total (including the size byte). The field may be set to 0.

`depth` takes one of the following values

0x0018 – images are in colour with no alpha

`width` and `height` are the maximum visual width and height of the stream described by this sample description, in pixels

12.1.4 Pixel aspect ratio and clean aperture

12.1.4.1 Definition

The pixel aspect ratio and clean aperture of the video may be specified using the `PixelAspectRatioBox` and `CleanApertureBox` sample entry boxes, respectively. These are both optional; if present, they override the declarations (if any) in structures specific to the video codec, which structures should be examined if these boxes are absent. For maximum compatibility, these boxes should follow, not precede, any boxes defined in or required by derived specifications.

The `PixelAspectRatioBox` is informative; if the decoded output of the codec is re-formatted to the dimensions in the track header, this will accomplish any needed adjustment to a uniformly-scaled grid.

In the `PixelAspectRatioBox`, `hSpacing` and `vSpacing` have the same units, but those units are unspecified: only the ratio matters. `hSpacing` and `vSpacing` may or may not be in reduced terms, and they may reduce to 1/1. Both of them shall be strictly positive.

NOTE 1 The pixel aspect ratio should not be confused with the picture aspect ratio, also known as the display aspect ratio, which is the ratio of the width to the height of the final displayed image (e.g. 16:9).

They are defined as the aspect ratio of a pixel, in arbitrary units. If a pixel appears H wide and V tall, then hSpacing/vSpacing is equal to H/V. This means that a square on the display that is n pixels tall needs to be n*vSpacing/hSpacing pixels wide to appear square.

There are notionally four values in the `CleanApertureBox`. These parameters are represented as a fraction N/D. The fraction may or may not be in reduced terms. We refer to the pair of parameters `fooN` and `fooD` as `foo`. For `horizOff` and `vertOff`, D shall be strictly positive and N may be positive or negative. For `cleanApertureWidth` and `cleanApertureHeight`, N shall be positive and D shall be strictly positive.

NOTE 2 These are fractional numbers for several reasons. First, in some systems the exact width after pixel aspect ratio correction is integral, not the pixel count before that correction. Second, if video is resized in the full aperture, the exact expression for the clean aperture might not be integral. Finally, because this is represented using centre and offset, a division by two is needed, and so half-values can occur.

Considering the pixel dimensions as defined by the VisualSampleEntry width and height. If picture centre of the image is at `pcX` and `pcY`, then `horizOff` and `vertOff` are defined as follows:

```
    pcX = horizOff + (width  - 1)/2

    pcY = vertOff  + (height - 1)/2;
```

Typically, `horizOff` and `vertOff` are zero, so the image is centred about the picture centre.

The leftmost/rightmost pixel and the topmost/bottommost line of the clean aperture fall at:

```
pcX ± (cleanApertureWidth - 1)/2
pcY ± (cleanApertureHeight - 1)/2;
```

The cropping implied by the `CleanApertureBox` is applied before any transformation defined by track or movie matrices.

12.1.4.2 Syntax

```
class PixelAspectRatioBox extends Box('pasp'){
   unsigned int(32) hSpacing;
   unsigned int(32) vSpacing;
}
class CleanApertureBox extends Box('clap'){
   unsigned int(32) cleanApertureWidthN;
   unsigned int(32) cleanApertureWidthD;

   unsigned int(32) cleanApertureHeightN;
   unsigned int(32) cleanApertureHeightD;


   unsigned int(32) horizOffN;
   unsigned int(32) horizOffD;


   unsigned int(32) vertOffN;
   unsigned int(32) vertOffD;

}
```

12.1.4.3 Semantics

`hSpacing`, `vSpacing`: define the relative width and height of a pixel;

`cleanApertureWidthN`, `cleanApertureWidthD`: a fractional number which defines the width of the clean aperture image

`cleanApertureHeightN`, `cleanApertureHeightD`: a fractional number which defines the height of the clean aperture image

`horizOffN`, `horizOffD`: a fractional number which defines the horizontal offset between the clean aperture image centre and the full aperture image centre. Typically 0.

`vertOffN`, `vertOffD`: a fractional number which defines the vertical offset between clean aperture image centre and the full aperture image centre. Typically 0.

12.1.5 Colour information

12.1.5.1 Definition

Colour information may be supplied in one or more `ColourInformationBox`es placed in a VisualSampleEntry. These should be placed in order in the sample entry starting with the most accurate (and potentially the most difficult to process), in progression to the least. These are advisory and concern rendering and colour conversion, and there is no normative behaviour associated with them; a reader may choose to use the most suitable. A `ColourInformationBox` with an unknown colour type may be ignored.

If used, an ICC profile may be a restricted one, under the code `'rICC'`, which permits simpler processing. That profile shall be of either the monochrome or three-component matrix-based class of input profiles, as defined by ISO 15076-1. If the profile is of another class, then the `'prof'` indicator shall be used.

If colour information is supplied in both this box, and also in the video bitstream, this box takes precedence, and over-rides the information in the bitstream.

NOTE When an ICC profile is specified, SMPTE RP 177[14] could be of assistance if there is a need to form the Y'CbCr to R'G'B' conversion matrix for the colour primaries described by the ICC profile.

12.1.5.2 Syntax

```
class ColourInformationBox extends Box('colr'){
   unsigned int(32) colour_type;
   if (colour_type == 'nclx')   /* on-screen colours */
   {
      unsigned int(16) colour_primaries;
      unsigned int(16) transfer_characteristics;
      unsigned int(16) matrix_coefficients;
      unsigned int(1)  full_range_flag;
      unsigned int(7)  reserved = 0;
   }
   else if (colour_type == 'rICC')
   {
      ICC_profile;   // restricted ICC profile
   }
   else if (colour_type == 'prof')
   {
      ICC_profile;   // unrestricted ICC profile
   }
}
```

12.1.5.3 Semantics

`colour_type`: an indication of the type of colour information supplied.

`colour_primaries` carries a **ColourPrimaries** value as defined in ISO/IEC 23091-2

`transfer_characteristics` carries a **TransferCharacteristics** value as defined in ISO/IEC 23091-2

`matrix_coefficients` carries a **MatrixCoefficients** value as defined in ISO/IEC 23091-2

`full_range_flag` carries a **VideoFullRangeFlag** as defined in ISO/IEC 23091-2

`ICC_profile`: an ICC profile as defined in ISO 15076-1 or ICC.1[13] is supplied.

12.1.6 Content light level

12.1.6.1 Definition

This box may be used to provide information about the light level in the content and may be present in a VisualSampleEntry. It is functionally equivalent to, and shall be as described in, the Content light level information SEI message in ITU-T H.265 | ISO/IEC 23008-2, with the addition that the provisions of CTA-861-G[35], in which zero in some cases codes an unknown value, may be used.

NOTE This is a `Box`, not a `FullBox` (similar to `PixelAspectRatioBox`).

12.1.6.2 Syntax

```
class ContentLightLevelBox extends Box('clli'){
   unsigned int(16) max_content_light_level;
   unsigned int(16) max_pic_average_light_level;
}
```

12.1.7 Mastering display colour volume

12.1.7.1 Definition

This box may be used to provide information about the colour primaries, white point, and mastering luminance in the content and may be present in a VisualSampleEntry. It is functionally equivalent to, and shall be as described in, the mastering display colour volume SEI message in ITU-T H.265 | ISO/IEC 23008-2, with the addition that the provisions of CTA-861-G[35] in which zero in some cases codes an unknown value may be used.

NOTE This is a `Box`, not a `FullBox` (similar to `PixelAspectRatioBox`).

12.1.7.2 Syntax

```
class MasteringDisplayColourVolumeBox extends Box('mdcv'){
   for (c = 0; c<3; c++) {
      unsigned int(16) display_primaries_x;
      unsigned int(16) display_primaries_y;
   }
   unsigned int(16) white_point_x;
   unsigned int(16) white_point_y;
   unsigned int(32) max_display_mastering_luminance;
   unsigned int(32) min_display_mastering_luminance;
}
```

12.1.8 Content colour volume

12.1.8.1 Definition

This box describes the colour volume characteristics of the associated pictures. These colour volume characteristics are expressed in terms of a nominal range, although deviations from this range may occur. It is functionally equivalent to, and shall be as described in, the content colour volume SEI message in Rec. ITU-T H.265 | ISO/IEC 23008-2 except that the box, in a sample entry, applies to the associated content and hence the initial two bits (corresponding to the ccv_cancel_flag and ccv_persistence_flag) take the value 0.

NOTE This is a `Box`, not a `FullBox` (similar to `PixelAspectRatioBox`).

12.1.8.2 Syntax

```
class ContentColourVolumeBox extends Box('cclv'){
   unsigned int(1) reserved1 = 0;    // ccv_cancel_flag
   unsigned int(1) reserved2 = 0;    // ccv_persistence_flag
   unsigned int(1) ccv_primaries_present_flag;
   unsigned int(1) ccv_min_luminance_value_present_flag;
   unsigned int(1) ccv_max_luminance_value_present_flag;
   unsigned int(1) ccv_avg_luminance_value_present_flag;
   unsigned int(2) ccv_reserved_zero_2bits = 0;
   if( ccv_primaries_present_flag ) {
      for( c = 0; c < 3; c++ ) {
         signed int(32) ccv_primaries_x[ c ];
         signed int(32) ccv_primaries_y[ c ];
      }
   }
   if( ccv_min_luminance_value_present_flag )
      unsigned int(32) ccv_min_luminance_value;
   if( ccv_max_luminance_value_present_flag )
      unsigned int(32) ccv_max_luminance_value;
   if( ccv_avg_luminance_value_present_flag )
      unsigned int(32) ccv_avg_luminance_value;
}
```

12.1.9 Ambient viewing environment

12.1.9.1 Definition

This box may be used to provide information about the characteristics of the nominal ambient viewing environment for the display of the associated video content and may be present in a `VisualSampleEntry`. The syntax elements of the ambient viewing environment box may assist the receiving system in adapting the received video content for local display in viewing environments that may be similar or may substantially differ from those assumed or intended when mastering the video content. It is functionally equivalent to, and shall be as described in, the ambient viewing environment SEI message in ITU-T H.265 |I ISO/IEC 23008-2.

NOTE This is a `Box`, not a `FullBox` (similar to `PixelAspectRatioBox`).

12.1.9.2 Syntax

```
class AmbientViewingEnvironmentBox extends Box('amve'){
   unsigned int(32) ambient_illuminance;
   unsigned int(16) ambient_light_x;
   unsigned int(16) ambient_light_y;
}
```

12.2 Audio media

12.2.1 Media handler

Audio media uses the `'soun'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.2.2 Sound media header

12.2.2.1 Definition

Box Types: `'smhd'`
Container: `MediaInformationBox` Box
Mandatory: Yes
Quantity: Exactly one specific media header shall be present

Audio tracks use the `SoundMediaHeaderBox` in the `MediaInformationBox` as defined in 8.4.5. The sound media header contains general presentation information, independent of the coding, for audio media. This header is used for all tracks containing audio.

12.2.2.2 Syntax

```
aligned(8) class SoundMediaHeaderBox
   extends FullBox('smhd', version = 0, 0) {
   template int(16) balance = 0;
   const unsigned int(16)   reserved = 0;
}
```

12.2.2.3 Semantics

`version` is an integer that specifies the version of this box

`balance` is a fixed-point 8.8 number that places mono audio tracks in a stereo space; 0 is centre (the normal value); full left is -1.0 and full right is 1.0.

12.2.3 Sample entry

12.2.3.1 Definition

Audio tracks use AudioSampleEntry or AudioSampleEntryV1.

The `samplerate`, `samplesize` and `channelcount` fields document the default audio output playback format for this media. The timescale for an audio track should be chosen to match the sampling rate, or be an integer multiple of it, to enable sample-accurate timing. When `channelcount` is a value greater than zero, it indicates the total number of channels in the audio stream.

The audio output format (`samplerate`, `samplesize` and `channelcount` fields) in the sample entry should be considered definitive only for codecs that do not record their own output configuration. If the audio codec has definitive information about the output format, it shall be taken as definitive; in this case the `samplerate`, `samplesize` and `channelcount` fields in the sample entry may be ignored, though sensible values should be chosen (for example, the highest possible sampling rate).

When it is desired to indicate an audio sampling rate greater than the value than can be represented in the `samplerate` field, then one of the following may be used:

1) If the system needs to rely on this signalling alone, e.g. because the codec does not itself provide the sample rate:

   — an `AudioSampleEntryV1` is used

   — a `SamplingRateBox` is present in the `AudioSampleEntryV1`, and it overrides the `samplerate` field and documents the actual sampling rate;

2) Otherwise:

   — a `SamplingRateBox` is present in the `AudioSampleEntryV1` or `AudioSampleEntry`, and it documents the actual sampling rate.

When a `SamplingRateBox` is present:

— the media timescale should be the same as the sampling rate, or an integer division or multiple of it;

— the `samplerate` field in the sample entry should contain a value that matches the media timescale left-shifted 16 bits (as for `AudioSampleEntry`), or be an integer division or multiple of it.

An `AudioSampleEntryV1` should only be used when needed; otherwise, for maximum compatibility, an `AudioSampleEntry` should be used. For maximum compatibility, the `SamplingRateBox`, `ChannelLayoutBox` and any DownMix and DRC boxes should follow, not precede, any boxes defined in or required by derived specifications.

Encoders should encode the DRC-related boxes in the `AudioSampleEntry` in the order given in 12.2.3.2. Decoders may ignore and discard the DRC-related boxes if they are not in that order. DRC-related boxes include `ChannelLayout`, `DownMixInstructions`, `DRCCoefficientsBasic`, `DRCInstructionsBasic`, `DRCCoefficientsUniDrc`, `DRCInstructionsUniDrc`, and `UniDrcConfigExtension`. The `DownMixInstructions` and `DRCInstructionsUniDrc` box cannot occur more than once if the box has `version`==1, but it can occur multiple times if `version`==0.

12.2.3.2 Syntax

```
class AudioSampleEntry(codingname) extends SampleEntry (codingname){
   const unsigned int(32)[2] reserved = 0;
   unsigned int(16) channelcount;
   template unsigned int(16) samplesize = 16;
   unsigned int(16) pre_defined = 0;
   const unsigned int(16) reserved = 0 ;
   template unsigned int(32) samplerate = { default samplerate of media}<<16;
   // optional boxes follow
   Box ();      // further boxes as needed
   ChannelLayout();
   DownMixInstructions() [];
   DRCCoefficientsBasic() [];
   DRCInstructionsBasic() [];
   DRCCoefficientsUniDRC() [];
   DRCInstructionsUniDRC() [];
   // we permit only one DRC Extension box:
   UniDrcConfigExtension();
   // optional boxes follow
   SamplingRateBox();
   ChannelLayout();
}
aligned(8) class SamplingRateBox extends FullBox('srat') {
   unsigned int(32) sampling_rate;
}
class AudioSampleEntryV1(codingname) extends SampleEntry (codingname){
   unsigned int(16) entry_version;   // shall be 1,
                  // and shall be in an stsd with version ==1
   const unsigned int(16)[3] reserved = 0;
   template unsigned int(16) channelcount;   // shall be correct
   template unsigned int(16) samplesize = 16;
   unsigned int(16) pre_defined = 0;
   const unsigned int(16) reserved = 0 ;
   template unsigned int(32) samplerate = 1<<16;
   // optional boxes follow
   SamplingRateBox();
   Box ();      // further boxes as needed
   ChannelLayout();
   DownMixInstructions() [];
   DRCCoefficientsBasic() [];
   DRCInstructionsBasic() [];
   DRCCoefficientsUniDRC() [];
   DRCInstructionsUniDRC() [];
   // we permit only one DRC Extension box:
   UniDrcConfigExtension();
   // optional boxes follow
   ChannelLayout();
}
```

12.2.3.3 Semantics

`channelcount` is the number of channels

0 — inapplicable/unknown

1 — mono

2 — stereo (left/right)

all other values — the codec configuration should identify the channel assignment.

`SampleSize` is in bits, and takes the default value of 16

`SampleRate` when a `SamplingRateBox` is absent is the sampling rate; when a `SamplingRateBox` is present, is a suitable integer multiple or division of the actual sampling rate. This 32-bit field is expressed as a 16.16 fixed-point number (hi.lo) `sampling_rate` is the actual sampling rate of the audio media in samples/second, expressed as a 32-bit integer

12.2.4 Channel layout

12.2.4.1 Definition

Box Types: 'chnl'
Container: Audio sample entry
Mandatory: No
Quantity: Zero or one

This box may appear in an audio sample entry to document the assignment of channels in the audio stream. It is recommended to use this box to convey the base channel count for the DownMixInstructions box and other DRC-related boxes specified in ISO/IEC 23003-4.

The channel layout can be all or part of a standard layout (from an enumerated list), or a custom layout (which also allows a track to contribute part of an overall layout).

A stream may contain channels, objects, neither, or both. A stream that is neither channel nor object structured can implicitly be rendered in a variety of ways.

12.2.4.2 Syntax

```
aligned(8) class ChannelLayout extends FullBox('chnl', version, flags=0) {   if
(version==0) {
      unsigned int(8) stream_structure;
      if (stream_structure & channelStructured) {
         unsigned int(8) definedLayout;
          if (definedLayout==0) {
            for (i = 1 ; i <= layout_channel_count ; i++) {
               //  layout_channel_count comes from the sample entry
               unsigned int(8) speaker_position;
               if (speaker_position == 126) {   // explicit position
                  signed int (16) azimuth;
                  signed int (8)  elevation;
               }
            }
         } else {
            unsigned int(64)   omittedChannelsMap;
                  // a '1' bit indicates 'not in this track'
         }
      }
      if (stream_structure & objectStructured) {
         unsigned int(8) object_count;
      }
   } else {
      unsigned int(4) stream_structure;
      unsigned int(4) format_ordering;
      unsigned int(8) baseChannelCount;
      if (stream_structure & channelStructured) {
         unsigned int(8) definedLayout;
         if (definedLayout==0) {
            unsigned int(8) layout_channel_count;
            for (i = 1 ; i <= layout_channel_count ; i++) {
               unsigned int(8) speaker_position;
               if (speaker_position == 126) {   // explicit position
                  signed int (16) azimuth;
                  signed int (8)  elevation;
               }
            }
         } else {
            int(4) reserved = 0;
            unsigned int(3) channel_order_definition;
            unsigned int(1) omitted_channels_present;
            if (omitted_channels_present == 1) {
                unsigned int(64)   omittedChannelsMap;
                    // a '1' bit indicates 'not in this track'
                }
            }
        }
        if (stream_structure & objectStructured) {
                    // object_count is derived from baseChannelCount
        }
    }
}
```

12.2.4.3 Semantics

`version` is an integer that specifies the version of this box (0 or 1). When authoring, version 1 should be preferred over version 0. Version 1 conveys the channel ordering, which is not always the case for version 0. Version 1 should be used to convey the base channel count for DRC.

`stream_structure` is a field of flags that define whether the stream has channel or object structure (or both, or neither); the following flags are defined, all other values are reserved:

1 the stream carries channels

2 the stream carries objects

`format_ordering` indicates the order of formats in the stream starting from the lowest channel index (see Table). Each format shall only use contiguous channel indices.

format_ordering	**Order**
0	unknown
1	Channels, possibly followed by Objects
2	Objects, possibly followed by Channels
Remaining values are reserved	-

`definedLayout` is a **ChannelConfiguration** from ISO/IEC 23091-3.

`speaker_position` is an **OutputChannelPosition** from ISO/IEC 23091-3. If an explicit position is used, then the azimuth and elevation are as defined as for speakers in ISO/IEC 23091-3. The channel order corresponds to the order of speaker positions.

`azimuth` is a signed value in degrees, as defined for **LoudspeakerAzimuth** in ISO/IEC 23091-3.

`elevation` is a signed value, in degrees, as defined for **LoudspeakerElevation** in ISO/IEC 23091-3.

`channel_order_definition` indicates where the ordering of the audio channels for the `definedLayout` are specified (see Table).

channel_order_definition	**Channel order specification**
0	as listed for the **ChannelConfigurations** in ISO/IEC 23091-3
1	Default order of audio codec specification
2	Channel ordering #2 of audio codec specification
3	Channel ordering #3 of audio codec specification
4	Channel ordering #4 of audio codec specification
Remaining values are reserved	-

`omitted_channels_present` is a flag that indicates if it is set to 1 that the `omittedChannelsMap` is present.

`omittedChannelsMap` is a bit-map of omitted channels; the bits in the channel map are numbered from least-significant to most-significant, and correspond in that ordering with the order of the channels for the configuration as documented in ISO/IEC 23091-3 ChannelConfiguration. 1-bits in the channel map mean that a channel is absent. A zero value of the map therefore always means that the given standard layout is fully present. The default value is 0.

`layout_channel_count` is the count of channels for the channel layout. The default value is 0 if `stream_structure` indicates that no channel structure is present. Otherwise, the value is the number of channels of the defined layout, if present, otherwise it is the value from the sample entry.

`object_count` is the count of channels that contain audio objects. The default value is 0. For version 1 and if the `objectStructured` flag is set, the value is computed as `baseChannelCount` minus the channel count of the channel structure.

`baseChannelCount` represents the combined channel count of the channel layout and the object count. The value must match the base channel count for DRC (see ISO/IEC 23003-4).

12.2.5 Downmix instructions

12.2.5.1 Definition

Box Types: `'dmix'`
Container: Audio sample entry
Mandatory: No
Quantity: Zero or more

The downmix can be controlled by the production facility if necessary. For instance, some content may require more attenuation of the surround channels before downmixing to maintain intelligibility.

The downmix support is designed so that any downmix (e.g. from 7.1 to quad as well as to stereo) can be described.

It is possible to declare the loudness characteristics of the signal after downmix, and after DRC and downmix.

If targetChannelCount*baseChannelCount is odd, the box is padded with 4 bits set to 0xF. The targetChannelCount shall be consistent with the targetLayout (if given), and shall be less than or equal to the `channelcount`.

Each downmix is uniquely identified by an ID.

In the following definition, `ceil()` is the ceiling function.

12.2.5.2 Syntax

```
aligned(8) class DownMixInstructions extends FullBox('dmix', version, flags=0) {
   if (version >= 1) {
      bit(1) reserved = 0;
      bit(7) downmix_instructions_count;
   } else {
      int downmix_instructions_count = 1;
   }
   for (a=1; a<=downmix_instructions_count; a++) {
      unsigned int(8) targetLayout;
       unsigned int(1) reserved = 0;
      unsigned int(7) targetChannelCount;
      bit(1) in_stream;
      unsigned int(7) downmix_ID;
      if (in_stream==0)
      {   // downmix coefficients are out of stream and supplied here
         int i, j;
         if (version >= 1) {
            bit(4) bs_downmix_offset;
            int size = 4;
            for (i=1; i <= targetChannelCount; i++){
               for (j=1; j <= baseChannelCount; j++) {
                  bit(5) bs_downmix_coefficient_v1;
                  size += 5;
               }
            }
            bit(8 ceil(size / 8) – size) reserved = 0; // byte align
         } else {
            for (i=1; i <= targetChannelCount; i++){
               for (j=1; j <= baseChannelCount; j++) {
                  bit(4) bs_downmix_coefficient;
               }
            }
         }
      }
   }
}
```

12.2.5.3 Semantics

`targetLayout` is a **ChannelConfiguration** from ISO/IEC 23091-3 and defines the resulting layout after downmix

`targetChannelCount` is the count of channels in the resulting stream, and shall correspond with the target layout

`downmix_ID` is an arbitrary value that identifies this downmix, and shall be unique among the DownMixInstructions in a given sample entry; there are two reserved values, 0 and 0x7F, which shall not be used

`version` is an integer that specifies the version of this box (0 or 1)

`bs_downmix_offset` is an offset in dB for all downmix coefficients that are defined in the `bs_downmix_coefficient_v1` field. It is encoded as defined in Table 7 using the following expression for:

Table 7 — Downmix offset encoding

Value [dB]	**Hex encoding (3 bits)**
0.0	0x0
	0x1
	0x2
reserved	other

`baseChannelCount` is the channel count of the audio signal in the base layout as also defined in ISO/IEC 23003-4. It should be derived from the `ChannelLayout` box, if present.

`in_stream` has a value of 1 when the downmix coefficients are in the stream. Otherwise, it is zero.

`bs_downmix_coefficient` is encoded as defined in Table 8 and Table 9:

Table 8 — Downmix coefficient encoding for non-LFE channel and version==0 (bs_downmix_coefficient)

Value	**Hex encoding (4 bits)**
0.00 dB	**0x0**
-0.50 dB	**0x1**
-1.00 dB	**0x2**
-1.50 dB	**0x3**
-2.00 dB	**0x4**
-2.50 dB	**0x5**
-3.00 dB	**0x6**
-3.50 dB	**0x7**

Table 8 *(continued)*

Value	Hex encoding (4 bits)
-4.00 dB	0x8
-4.50 dB	0x9
-5.00 dB	0xA
-5.50 dB	0xB
-6.00 dB	0xC
-7.50 dB	0xD
-9.00 dB	0xE
-∞ dB	0xF

Table 9 — Downmix coefficient encoding for LFE channel and version==0 (bs_downmix_coefficient)

Value	Hex encoding (4 bits)
10.00 dB	0x0
6.00 dB	0x1
4.5 dB	0x2
3.00 dB	0x3
1.50 dB	0x4
0.00 dB	0x5
-1.50 dB	0x6
-3.00 dB	0x7
-4.50 dB	0x8
-6.00 dB	0x9
-10.00 dB	0xA
-15.00 dB	0xB
-20.00 dB	0xC
-30.00 dB	0xD
-40.00 dB	0xE
-∞ dB	0xF

`bs_downmix_coefficient_v1` is encoded as defined in Table 10:

Table 10 — Downmix coefficient encoding for version>=1 (bs_downmix_coefficient_v1)

Value	Hex encoding (5 bits)
10.00 dB	0x00
6.00 dB	0x01
4.50 dB	0x02
3.00 dB	0x03
1.50 dB	0x04
0.00 dB	0x05
-0.50 dB	0x06
-1.00 dB	0x07
-1.50 dB	0x08
-2.00 dB	0x09
-2.50 dB	0x0A

Table 10 *(continued)*

Value	Hex encoding (5 bits)
-3.00 dB	0x0B
-3.50 dB	0x0C
-4.00 dB	0x0D
-4.50 dB	0x0E
-5.00 dB	0x0F
-5.50 dB	0x10
-6.00 dB	0x11
-6.50 dB	0x12
-7.00 dB	0x13
-7.50 dB	0x14
-8.00 dB	0x15
-9.00 dB	0x16
-10.00 dB	0x17
-11.00 dB	0x18
-12.00 dB	0x19
-15.00 dB	0x1A
-20.00 dB	0x1B
-25.00 dB	0x1C
-30.00 dB	0x1D
-40.00 dB	0x1E
-∞ dB	0x1F

12.2.6 DRC information

A DRC is used in the encoder to generate gain values using one of the pre-defined DRC characteristics as defined in ISO/IEC 23091-3 or a characteristic defined in ISO/IEC 23003-4. The coefficients are placed either in-stream or in an associated meta-data track. Alternatively, coefficients are generated at the decoder based on transmitted parametric DRC configurations.

For some content, such as some multi-channel content, it may be advantageous to use different DRC characteristics in different channels. For instance, if speech is exclusively present in the center channel, this feature can be very useful. It is supported by the assignment of DRC characteristics to audio channels.

It is possible to declare the loudness characteristics of the signal after DRC.

DRC support includes supporting in-stream DRC coefficients, and a separate track carrying them; the latter is particularly useful for legacy coding systems (including uncompressed audio) that have no provision for in-stream coefficients.

In the ISO base media file format, the audio content may be carried in multiple tracks where a base track contains the DRC metadata for all tracks. The additional tracks are referenced by the base track using a track reference of type `'adda'` (additional audio). The channels processed by the DRC are all the channels in the base track, plus all the channels in track(s) referenced, in the order of the references. The DRC channel groups apply to all those channels (even if they are channels in a track that is disabled or not currently being played).

The boxes `DRCCoefficientsBasic`, `DRCCoefficientsUniDRC`, `DRCInstructionsBasic`, `DRCInstructionsUniDRC`, and `UniDrcConfigExtension` may occur in an AudioSampleEntry and are defined in ISO/IEC 23003-4.

12.2.7 Audio stream loudness

12.2.7.1 Definition

Box Types: `'ludt'`
Container: `UserDataBox` of the corresponding `TrackBox`
Mandatory: No
Quantity: Zero or more

Loudness declarations are placed in `UserDataBox`es, to enable their presence and update in movie fragments. In particular, in live scenarios, user-data in the initial `MovieBox` may be a 'promise not to exceed' or 'best guess', and then user-data updates give better (but still generally valid) values. Thus, for example, a loudness range in this user data that is associated with a particular set of DRC instructions constitutes a 'promise' rather than a measurement, under these circumstances.

Several metadata values are available that describe aspects of the dynamic range. The size of the dynamic range can be useful in adjusting the DRC characteristic, e.g. the DRC is less aggressive if the dynamic range is small or the DRC can even be turned off.

True Peak and maximum loudness values can be useful for estimating the headroom, for instance when loudness normalization results in a positive gain [dB] or when headroom is needed to avoid clipping of the downmix. The DRC characteristic can then be adjusted to approach a headroom target. The peak level of the associated content is represented here in a coding-independent way.

The audio sound pressure level that the content was mixed to can also be documented. (If audio is listened to at a level other than the mixing level, this can affect the perceived tonal balance.)

The following measures may also be used:

— Maximum of the loudness range derived from EBU-Tech 3342 [26]

— Maximum momentary loudness derived from ITU-R BS.1771-1 [24] or EBU-Tech 3341 [25]

— Maximum short-term loudness derived from ITU-R BS.1771-1 [24] or EBU-Tech 3341 [25]

— Short-term loudness defined in ITU-R BS.1771-1[24] or EBU-Tech 3341[25]

Under some circumstances it can be desirable to indicate the loudness characteristics of an album, in each song that the album contains. A separate box can be specified for that purpose. The TrackLoudnessInfo and AlbumLoudnessInfo provide loudness information for the song, and for the entire album which contains the song, respectively.

The program loudness shall be measured using ITU-R BS.1770-4 over the associated content; the 'anchor loudness' is the loudness of the anchor content, where what that content is, is determined by the content author; one suitable value (especially for content for which the main content is speech) is 'dialog normal level' or DialNorm as defined in ATSC Doc. A/52:2012[27]. ISO/IEC 23003-4 specifies the measurement systems, measurement methods and the coding of all loudness and peak-related values.

12.2.7.2 Syntax

```
aligned(8) class LoudnessBaseBox extends FullBox(loudnessType, version, flags=0) {
   if (version >= 2) {
      unsigned int(2) loudness_info_type;
      unsigned int(6) loudness_base_count;
      if (loudness_info_type == 1 || loudness_info_type == 2) {
         unsigned int(1) reserved = 0;
         unsigned int(7) mae_group_ID;
      }
      else if (loudness_info_type == 3) {
         unsigned int(3) reserved = 0;
         unsigned int(5) mae_group_preset_ID;
      }
   }
   else if (version == 1) {
      unsigned int(2) reserved = 0;
      unsigned int(6) loudness_base_count;
   } else {
      int loudness_base_count = 1;
   }
   for (a=1; a<=loudness_base_count; a++) {
      if (version >= 1) {
         unsigned int(2) reserved = 0;
         unsigned int(6) EQ_set_ID;      // to match an EQ box
      }
      unsigned int(3) reserved = 0;
      unsigned int(7) downmix_ID;      // matching downmix
      unsigned int(6) DRC_set_ID;      // to match a DRC box
      signed int(12)  bs_sample_peak_level;
      signed int(12)  bs_true_peak_level;
      unsigned int(4) measurement_system_for_TP;
      unsigned int(4) reliability_for_TP;
      unsigned int(8) measurement_count;
      int i;
      for (i = 1 ; i <= measurement_count; i++){
         unsigned int(8) method_definition;
         unsigned int(8) method_value;
         unsigned int(4) measurement_system;
         unsigned int(4) reliability;
      }
   }
}
aligned(8) class TrackLoudnessInfo extends LoudnessBaseBox('tlou') { }
aligned(8) class AlbumLoudnessInfo extends LoudnessBaseBox ('alou') { }
aligned(8) class LoudnessBox extends Box('ludt') {
   // not more than one TrackLoudnessInfo box with version>=1 is allowed
   loudness         TrackLoudnessInfo[];
   // not more than one AlbumLoudnessInfo box with version>=1 is allowed   albumLoudness
AlbumLoudnessInfo[];
}
```

12.2.7.3 Semantics

`version` is an integer that specifies the version of this box (0, 1 or 2)

`loudness_info_type` is the type of audio scene described by the loudness information. It shall take a value of zero unless other types are supported by the loudness processing. For defined values refer to the corresponding loudnessInfoType specification in ISO/IEC 23008-3[31].

`mae_group_ID` is a unique identifier for a group of metadata elements as specified in ISO/IEC 23008-3[31].

`mae_group_preset_ID` is a unique identifier for a group preset as specified in ISO/IEC 23008-3[31].

`downmix_ID` when zero, declares the loudness characteristics of the layout without downmix. If non-zero, this box declares the loudness after applying the downmix with the matching `downmix_ID` and shall match a value in exactly one box in the sample entry of this track

`DRC_set_ID` when zero, declares the characteristics without applying a DRC. If non-zero, this box declares the loudness after applying the DRC with the matching `DRC_set_ID` and shall match a value in exactly one box in the sample entry of this track

`EQ_set_ID` when zero, declares the characteristics without applying EQ. If non-zero, this box declares the loudness after applying the EQ with the matching `EQ_set_ID` and shall match a value in exactly one box in the `UniDrcConfigExtension` of this track

`bs_sample_peak_level` takes a value for the sample peak level as defined in ISO/IEC 23003-4; all other values are reserved

`bs_true_peak_level` takes a value for the true peak level as defined in ISO/IEC 23003-4; all other values are reserved `measurement_system_for_TP` takes an index for the measurement system as defined in ISO/IEC 23003-4; all other values are reserved

`method_definition` takes an index for the measurement method as defined in ISO/IEC 23003-4; all others are reserved

`measurement_system` takes an index for the measurement system as defined in ISO/IEC 23003-4; all others are reserved

`reliability` and `reliability_for_TP` each take one of the following values (all other values are reserved):

0: Reliability is unknown

1: Value is reported/imported but unverified

2: Value is a ‘not to exceed’ ceiling

3: Value is measured and accurate

12.3 Metadata media

12.3.1 Media handler

Timed metadata media uses the `'meta'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

NOTE 1 MPEG-7 streams, which are a specific kind of metadata stream, have their own handler declared, documented in the MP4 file format (ISO/IEC 14496-14[22]).

NOTE 2 metadata tracks are linked to the track they describe using a track-reference of type `'cdsc'`.

12.3.2 Media header

Metadata tracks use a `NullMediaHeaderBox`, as defined in subclause 8.4.5.2.

12.3.3 Sample entry

12.3.3.1 Definition

Timed metadata tracks use MetaDataSampleEntry.

In case of XML metadata a `BitRateBox` in the SampleEntry can be used to choose the appropriate memory representation format (DOM, STX).

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.

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.

12.3.3.2 Syntax

```
class MetaDataSampleEntry(codingname) extends SampleEntry (codingname) {
}
class XMLMetaDataSampleEntry() extends MetaDataSampleEntry ('metx') {
   utf8string content_encoding; // optional
   utf8list namespace;
   utf8list schema_location; // optional
}
class TextConfigBox() extends Fullbox ('txtC', 0, 0) {
   utf8string text_config;
}
class TextMetaDataSampleEntry() extends MetaDataSampleEntry ('mett') {
   utf8string content_encoding; // optional
   utf8string mime_format;
   TextConfigBox (); // optional
}
class MIMEBox() extends Fullbox ('mime', 0, 0) {
   utf8string content_type;
}
aligned(8) class URIBox extends FullBox('uri ', version = 0, 0) {
   utf8string 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
}
```

12.3.3.3 Semantics

`content_encoding` provides a MIME type which identifies the content encoding of the timed metadata. It is defined in the same way as for an `ItemInfoEntry` in this document. If not present (an empty string is supplied) the timed metadata is not encoded. An example for this field is ‘application/zip’. Note that no MIME types for BiM [ISO/IEC 23001-1] and TeM [ISO/IEC 15938-1] currently exist. Thus, the experimental MIME types ‘application/x-BiM’ and ‘text/x-TeM’ shall be used to identify these encoding mechanisms.

`namespace` provides one or more XML namespaces to which the sample documents conform. When used for metadata, this is needed for identifying its type, e.g. gBSD or AQoS [MPEG-21-7] and for decoding using XML aware encoding mechanisms such as BiM.

`schema_location` provides zero or more URLs for XML schema(s) to which the sample document conforms. If there is one namespace and one schema, then this field shall be the URL of the one schema. If there is more than one namespace, then the syntax of this field shall adhere to that for xsi:schemaLocation attribute as defined by XML. When used for metadata, this is needed for decoding of the timed metadata by XML aware encoding mechanisms such as BiM.

`mime_format` provides a MIME type which identifies the content format of the samples. Examples for this field include ‘text/html’ and ‘text/plain’.

`text_config` provides the initial text of each document which is prepended before the contents of each sync sample.

`content_type` is a string corresponding to the MIME type each XML document carried in the stream would have if it were delivered on its own, possibly including sub-parameters.

NOTE This implies that if two XML documents carried in the same track have different MIME types (or sub-parameters), each document needs to be associated with a different sample entry.

`theURI` is a URI formatted according to the rules in 6.3.3;

`uri_initialization_data` is opaque data whose form is defined in the documentation of the URI form.

12.4 Hint media

12.4.1 Overview

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 in 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 shall 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 `HandlerBox`

— They use a `HintMediaHeaderBox`

— They use a `HintSampleEntry` in the `SampleDescriptionBox`, 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 shall not use composition time offsets. The process of hinting computes transmission times correctly as the presentation time derived from the decoding time.

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.

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.

12.4.2 Media handler

Hint media uses the `'hint'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.4.3 Hint media header

12.4.3.1 Hint media header box

Box Types: `'hmhd'`
Container: `MediaInformationBox`
Mandatory: Yes
Quantity: Exactly one specific media header shall be present

Hint tracks use the `HintMediaHeaderBox` in the `MediaInformationBox`, as defined in 8.4.5. The hint media header contains general information, independent of the protocol, for hint tracks. (A PDU is a protocol data unit.)

12.4.3.2 Syntax

```
aligned(8) class HintMediaHeaderBox
   extends FullBox('hmhd', version = 0, 0) {
   unsigned int(16)   maxPDUsize;
   unsigned int(16)   avgPDUsize;
   unsigned int(32)   maxbitrate;
   unsigned int(32)   avgbitrate;
   unsigned int(32)   reserved = 0;
}
```

12.4.3.3 Semantics

`version` is an integer that specifies the version of this box

`maxPDUsize` gives the size in bytes of the largest PDU in this (hint) stream

`avgPDUsize` gives the average size of a PDU over the entire presentation

`maxbitrate` gives the maximum rate in bits/second over any window of one second

`avgbitrate` gives the average rate in bits/second over the entire presentation

12.4.4 Sample entry

12.4.4.1 Definition

Hint tracks use an entry format specific to their protocol, with an appropriate name.

For hint tracks, the sample description contains appropriate declarative data for the streaming protocol being used, and the format of the hint track. The definition of the sample description is specific to the protocol.

The ‘protocol’ and ‘codingname’ fields are registered identifiers that uniquely identify the streaming protocol or compression format decoder to be used. A given protocol or codingname may have optional or required extensions to the sample description (e.g. codec initialization parameters). All such extensions shall be within boxes; these boxes occur after the required fields. Unrecognized boxes shall be ignored.

12.4.4.2 Syntax

```
class HintSampleEntry() extends SampleEntry (protocol) {
}
```

12.5 Text media

12.5.1 Media handler

The timed text media type indicates that the associated decoder will process only text data. Timed text media uses the `'text'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.5.2 Media header

Timed text tracks use a `NullMediaHeaderBox`, as defined in subclause 8.4.5.2.

12.5.3 Sample entry

12.5.3.1 Definition

Timed text tracks use PlainTextSampleEntry.

12.5.3.2 Syntax

```
class PlainTextSampleEntry(codingname) extends SampleEntry (codingname) {
}
class SimpleTextSampleEntry extends PlainTextSampleEntry ('stxt') {
   utf8string   content_encoding;   // optional
   utf8string   mime_format;
   TextConfigBox ();                 // optional
}
```

12.5.3.3 Semantics

`content_encoding` provides a MIME type which identifies the content encoding of the timed text. It is defined in the same way as for an `ItemInfoEntry` in this document. If not present (an empty string is supplied) the timed text is not encoded. An example for this field is 'application/zip'.

`mime_format` provides a MIME type which identifies the content format of the samples. Examples for this field include 'text/html' and 'text/plain'.

12.6 Subtitle media

12.6.1 Media handler

The subtitle media type indicates that the associated decoder will process text data and possibly images. Subtitle media uses the `'subt'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.6.2 Subtitle media header

12.6.2.1 Definition

Subtitle tracks use the `SubtitleMediaHeaderBox` in the `MediaInformationBox`, as defined in 8.4.5. The subtitle media header contains general presentation information, independent of the coding, for subtitle media. This header is used for all tracks containing subtitles.

12.6.2.2 Syntax

```
aligned(8) class SubtitleMediaHeaderBox
   extends FullBox ('sthd', version = 0, flags = 0){
}
```

12.6.2.3 Semantics

`version` - is an integer that specifies the version of this box.

`flags` - is a 24-bit integer with flags (currently all zero).

12.6.3 Sample entry

12.6.3.1 Definition

Subtitle tracks use SubtitleSampleEntry.

12.6.3.2 Syntax

```
class SubtitleSampleEntry(codingname) extends SampleEntry (codingname) {
}
class XMLSubtitleSampleEntry() extends SubtitleSampleEntry ('stpp') {
   utf8list namespace;
   utf8list schema_location; // optional
   utf8list auxiliary_mime_types;
            // optional, required if auxiliary resources are present
}
class TextSubtitleSampleEntry() extends SubtitleSampleEntry ('sbtt') {
   utf8string content_encoding; // optional
   utf8string mime_format;
   TextConfigBox (); // optional
}
```

12.6.3.3 Semantics

`content_encoding` provides a MIME type which identifies the content encoding of the subtitles. It is defined in the same way as for an `ItemInfoEntry` in this document. If not present (an empty string is supplied) the subtitle samples are not encoded. An example for this field is 'application/zip'.

`namespace` is one or more XML namespaces to which the sample documents conform. When used for metadata, this is needed for identifying its type, e.g. gBSD or AQoS [MPEG-21-7] and for decoding using XML aware encoding mechanisms such as BiM.

`schema_location` is zero or more URLs for XML schema(s) to which the sample document conforms. If there is one namespace and one schema, then this field shall be the URL of the one schema. If there is more than one namespace, then the syntax of this field shall adhere to that for xsi:schemaLocation attribute as defined by XML. When used for metadata, this is needed for decoding of the timed metadata by XML aware encoding mechanisms such as BiM.

`mime_format` provides a MIME type which identifies the content format of the samples. Examples for this field include 'text/html' and 'text/plain'.

`auxiliary_mime_types` indicates the media type of all auxiliary resources, such as images and fonts, if present, stored as subtitle sub-samples.

`optional_box` may be a `BitRateBox` or a `MIMEBox` or other box. When both `BitRateBox` and `MIMEBox` are present, they may be in any order. Parsers shall allow other boxes to be present.

12.7 Font media

12.7.1 Media handler

Font media uses the `'fdsm'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.7.2 Media header

Font tracks use a `NullMediaHeaderBox`.

12.7.3 Sample entry

12.7.3.1 Definition

Font streams use a `FontSampleEntry`.

12.7.3.2 Syntax

```
class FontSampleEntry(codingname) extends SampleEntry (codingname){
   //other boxes from derived specifications
}
```

12.8 Transformed media

12.8.1 General

Protected media is described in 8.12.

Incomplete media is described in 8.17.

Restricted media is described in 8.15.

12.8.2 Multiple transformations for a single transformed media track

A transformed media track may have undergone several transformations of different types and shall not have undergone more than one transformation of any particular type.

NOTE For example, a transformed track could be both protected and restricted.

The following process applies to conclude the untransformed sample entry type of a transformed media track:

1) Let schemeInfoContainerBox be `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or `CompleteTrackInfoBox` when the track sample entry type indicates an encrypted, restricted, or incomplete media track, respectively.

2) Let dataFormat be equal to the `data_format` value of `OriginalFormatBox` of schemeInfoContainerBox.

3) If dataFormat indicates a transformed media track, schemeInfoContainerBox is updated to be `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or `CompleteTrackInfoBox` when dataFormat indicates an encrypted, restricted, or incomplete media track, respectively. The process continues from step 2.

4) Otherwise (dataFormat does not indicate a transformed media track), the untransformed sample entry type is concluded to be equal to dataFormat.

12.8.3 Determining the untransformed sample entry type

A transformed media track may have undergone several transformations of different types but cannot have undergone more than one transformation of any particular type.

NOTE For example, a transformed track could be both protected and restricted, in which case the relevant boxes in the sample entry could be set as follows:

```
SampleEntry('encv') {
   ProtectionSchemeInfoBox {
      OriginalFormatBox;   // data_format is 'resv'
      SchemeTypeBox;
      SchemeInformationBox;
   }
   RestrictedSchemeInfoBox {
      OriginalFormatBox; // data_format indicates a codec, e.g. 'avc1'
      SchemeTypeBox;
      SchemeInformationBox;
   }
   // Boxes specific to the untransformed sample entry type
   // For 'avc1', these would include AVCConfigurationBox
}
```

The following process applies to conclude the untransformed sample entry type (as defined in the following subclause) of a transformed media track:

1) Let schemeInfoContainerBox be any `ProtectionSchemeInfoBox`, the `RestrictedSchemeInfoBox`, or the `CompleteTrackInfoBox` when the sample entry type of the track (i.e., the `format` value of a `SampleEntry` directly contained in the `SampleDescriptionBox`) indicates an encrypted, restricted, or incomplete media track, respectively.

2) Let dataFormat be equal to the `data_format` value of `OriginalFormatBox` of schemeInfoContainerBox.

3) If dataFormat indicates a transformed media track, schemeInfoContainerBox is updated to be `ProtectionSchemeInfoBox`, `RestrictedSchemeInfoBox`, or `CompleteTrackInfoBox` when dataFormat indicates an encrypted, restricted, or incomplete media track, respectively. The process continues from step 2.

4) Otherwise (dataFormat does not indicate a transformed media track), the untransformed sample entry type is concluded to be equal to dataFormat.

12.8.4 The `'codecs'` MIME parameter for a transformed media track

The 'codecs' parameter value for transformed media tracks is described in Annex K.

12.9 Multiplexed timed metadata tracks

12.9.1 General

Multiplexed timed metadata tracks

a) allow the carriage of any user-data item (untimed metadata) as timed metadata; this allows the 'radio station' case where song title, performer, etc. need to change as songs change;

b) allow the multiplexing together of formats; so, for example, a camera might record both location and facing direction in the same sample.

12.9.2 Overall design

Timed metadata multiplex is a specific format of a metadata track; it has a sample entry code, sample entry definitions, and a sample format. The sample entry code is `'mebx'`, standing for 'metadata boxed'.

The usual track reference(s) are used (notably `'cdsc'`).

The sample entry sets up an association between a compact, fixed-size, identifier used in the streams (repeatedly) and the permanent, possibly long-format, identifier, of the metadata and any parameters.

12.9.3 Sample format

An access unit (also known as media sample) is structured as a concatenation of one or more value `Box`es (as defined in subclause 4.2). The four-character-codes of these boxes are local keys that are declared in the sample entry; the setup for these keys allows various associations (e.g. as equivalent to a user-data item, as equivalent to an un-multiplexed timed metadata sample, etc.). The content of the box is defined by the association.

If no value for a particular key is present in the access unit at the given time, the interpretation should be that there is no metadata of that type at the time. Metadata values for that key for other times (e.g., from a previous access unit) should not be interpreted as applying to the target time.

Metadata access units are sync samples (also known as “I-frames” in video) when all the contained boxes in that sample would have been sync samples if they had been in their own sample, un-multiplexed.

If no values for any key are present for a time range, the best practice is to include an access unit with only a “NULL” data entry for the time range as defined in subclause 12.9.5. An empty track edit list entry could be used to indicate there is no metadata for a range of movie time.

12.9.4 Sample entry format

12.9.4.1 General

The sample entry for boxed timed metadata is the `BoxedMetadataSampleEntry`:

```
aligned(8) class BoxedMetadataSampleEntry
   extends MetadataSampleEntry ('mebx') {
   MetadataKeyTableBox();            // mandatory
   BitRateBox ();                     // optional
}
```

The only required box within `BoxedMetadataSampleEntry` is `MetadataKeyTableBox` which defines what metadata values may be found in the AUs of the track.

`MetadataKeyTableBox` (defined in 12.9.4.2) is a table indicating the set of keys and information about each key that may occur in associated access units.

`BitRateBox` is an optional box to signal the bitrate of the metadata stream.

12.9.4.2 Metadata key table box

Box Type: `'keys'`
Container: `BoxedMetadataSampleEntry`
Mandatory: Yes
Quantity: Exactly one

The `MetadataKeyTableBox` contains a table of keys and mappings to payload data in the corresponding access units. It is defined as:

```
aligned(8) class MetadataKeyTableBox extends Box('keys') {
   MetadataKeyBox[];
}
```

This is a box containing one or more instances of `MetadataKeyBox`, one for each “configuration” of key that may occur in the access units of the track. For example, if there are two keys, there will be two `MetadataKeyBox` boxes in the `MetadataKeyTableBox` – one for each key.

If the `MetadataKeyTableBox` does not contain a key for which a client is searching and there is no `MetadataInlineKeysPresentBox` signaling the possible presence of inline keys, no access units associated with this sample entry contain values with that key.

If the MetadataKeyTableBox does contain a particular key, this does not however guarantee that any access units containing a value for the key were written. Clients finding a key in the MetadataKeyTableBox may still need to look through the track's access units for values to determine if the track has the particular metadata.

NOTE This rule allows a sample entry to be populated (say during a capture process) with keys that might be discovered and then access units to be written with a binding only for the keys found. If never used, there's no requirement that the sample entry be rewritten to exclude the key that wasn't needed. This makes writing using movie fragments easier as the sample entries in the initial movie never need to be rewritten.

If it is possible to remove unused sample entries efficiently and rewrite the sample entry, this is preferred.

12.9.4.3 Metadata key box

12.9.4.3.1 Definition

Box Type: locally defined
Container: MetadataKeyTableBox
Mandatory: Yes
Quantity: One or more

The box type for each MetadataKeyBox is here referred to as 'local_key_id' and serves (1) as a unique identifier among all MetadataKeyBoxes and (2) as the identifier for the metadata value boxes within access units that have that key.

The box type for the contained MetadataKeyBox is 'local' to the containing track and corresponds to the box types (32-bit integers or four-character-codes) for boxes within metadata access units that hold that particular metadata value. For example, if the MetadataKeyBox has the box type of 'stuf', any boxes of type 'stuf' in access units sharing this sample entry hold the value for this key. Any value other than 0 and 0xFFFFFFFF fitting in a 32-bit big endian integer can be used (e.g., 'stuf', the integer 72) but it is recommended that it be mnemonic if possible.

There are two reserved box types for boxes of type MetadataKeyBox. A local_key_id of 0 in the MetadataKeyBox indicates that the MetadataKeyBox is unused and should not be interpreted. This allows the key to be marked as unused in the sample entry without requiring the sample entry and parent atoms to be rewritten/resized. (A box-type of 0 in a sample indicates 'null' metadata, see below). A local_key_id of 0xFFFFFFFF should never occur in MetadataKeyTableBox as this special value can be used to signal inline key/value boxes within access units.

All other box types are available for use.

NOTE Because the children boxes within MetadataKeyTableBox can take on any box type, there is be no special interpretation of the box type for contained boxes other than the special value 0. Therefore, including a FreeSpaceBox does not have the conventional meaning in the MetadataKeyBox. Even so, it is suggested that the use of overly confusing use of existing four-character-codes be avoided.

Each MetadataKeyBox contains a variable number of boxes that define the key structure, the datatype for values, optionally the locale for the values, and optional setup information needed to interpret the value. In future versions of this document, new children boxes may be introduced.

12.9.4.3.2 Syntax

```
aligned(8) class MetadataKeyBox extends Box(local_key_id) {
   MetadataKeyDeclarationBox();
   MetadataLocaleBox();                // optional
   MetadataSetupBox();                   // optional
   // other boxes may be present
}
```

12.9.4.4 Metadata key declaration box

12.9.4.4.1 Definition

Box Type: `'keyd'`
Container: `MetadataKeyBox`
Mandatory: Yes
Quantity: Exactly one

The `MetadataKeyDeclarationBox` holds the key namespace and key value of that namespace for the given values.

12.9.4.4.2 Syntax

```
aligned(8) class MetadataKeyDeclarationBox extends Box('keyd') {
   unsigned int(32)   key_namespace;
   unsigned int(8)   key_value[];
}
```

12.9.4.4.3 Semantics

`key_namespace` is a 32-bit identifier describing the domain and the structure of the `key_value`. For example, this could indicate that `key_value` is a reverse-address style null-terminated string using UTF-8 (e.g., "com.foo.mymetadata"), a binary four-character codes (e.g., `'cprt'` user data key), a Uniform Resource Identifier (URI), or other structures (e.g., native formats from metadata standards such as MXF). New `key_namespace`s are registered, but as a URI can often be used, using the URI key namespace may be sufficient for most uses. Unrecognized key namespaces shall cause the associated key, and the sample data for that key, data to be ignored.

`key_value` is an array of bytes holding the key and whose interpretation is defined by the associated `key_namespace` field.

12.9.4.5 Metadata locale box

12.9.4.5.1 Definition

Box Type: `'loca'`
Container: `MetadataKeyBox`
Mandatory: No
Quantity: Zero or one

A metadata value may optionally be tagged with its locale so that it may be chosen based upon the user's language, country, etc. This makes it possible to include several keys of the same key type (e.g., copyright or scene description) but with differing locales for users of different languages or locations. This allows one track to hold different localizations for a particular key.

This is accomplished by including a `MetadataLocaleBox` within the `MetadataKeyBox`.

If the `MetadataLocaleBox` is absent, corresponding metadata values should be considered appropriate for all locales. Example locale strings include 'en-US', 'fr-FR', or 'zh-CN'.

12.9.4.5.2 Syntax

```
aligned (8) class MetadataLocaleBox extends Box('loca') {
   utf8string locale_string;
}
```

12.9.4.5.3 Semantics

`locale_string` is a nul-terminated string of UTF-8 characters (i.e., a "C string") holding a language tag complying with IETF BCP 47.

12.9.4.6 Metadata setup box

12.9.4.6.1 Definition

Box Type: 'setu'
Container: MetadataKeyBox
Mandatory: No
Quantity: Zero or one

For some key_namespaces, setup data is needed. The contents of the MetadataSetupBox are defined by the key_namespace. (For example, it contains the boxes that would have been in the sample entry, for the case of multiplexed timed metadata).

12.9.4.6.2 Syntax

```
aligned(8) class MetadataSetupBox extends Box('setu') {
   // leaf data or array of Boxes
}
```

12.9.5 Defined formats

12.9.5.1 Null

A NULL value (and hence access unit) can be signalled using a single box with the reserved value of 0 for local_key_id. The contents of any such boxes should be ignored and may be present. No setup is needed for NULL values; they may be present in any multiplex without declaration.

Null values may also be used to pad samples e.g. to a constant size, or to mark a value as no longer relevant without having to edit the file or samples, simply by changing the box type to 0.

NULL values would be sync samples if they occurred in their own sample, un-multiplexed.

12.9.5.2 User-data

The key_namespace 'uiso' (ISO user-data) indicates that the key is mapped to the user-data four-character-code indicated by the key_value, which shall be 4 bytes in big-endian order, and shall be a code valid as user-data. There is no other setup data for this namespace (no MetadataSetupBox).

The value box is exactly the box that would have occurred in the UserDataBox but using the local_key_id.

For ease of reading and inspection, it is suggested that the local_key_id and the original user-data four-character-code be equal. However readers shall not rely on any particular values of local_key_id.

User-data items would be sync samples if they occurred in their own sample, un-multiplexed.

12.9.5.3 Un-multiplexed timed metadata

The key_namespace 'me4c' (metadata sample entry four-character-code) indicates that the values could have been timed metadata samples in their own right.

This key_namespace indicates that the key is mapped to the sample-entry four-character-code, that is valid for metadata tracks, indicated by the key_value, which shall be 4 bytes in big-endian order, and shall be a code valid for a metadata track sample entry. All the extra data that would have been present as mandatory extensions to the MetaDataSampleEntry for that sample entry four-character-code shall be present in the MetadataSetupBox.

The value box is a simple Box whose content is exactly the sample that would have occurred in an un-multiplexed context.

For ease of reading and inspection, it is suggested that the `local_key_id` and the original sample-entry four-character-code be equal. However readers shall not rely on any particular values of `local_key_id`.

NOTE These are already defined by this document, and there are others defined elsewhere (e.g. 3GPP): `XMLMetaDataSampleEntry`, `TextMetaDataSampleEntry` and `URIMetaSampleEntry`.

12.9.5.4 Other forms

Other `key_namespace`s may be registered and used. If an unrecognized `key_namespace` is found, the associated sample entry and sample data shall be ignored.

Other possible `key_namespace`s include URI and URNs, though formats for timed metadata (and hence that could be multiplexed) for URIs already exist.

12.10 Volumetric visual media

12.10.1 Media handler

Volumetric visual media uses the `'volv'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3. Multiple volumetric visual tracks may be present in the file.

12.10.2 Media header

12.10.2.1 Definition

Box Type: `'vvhd'`
Container: `MediaInformationBox`
Mandatory: Yes
Quantity: Exactly one

Volumetric visual tracks use the `VolumetricVisualMediaHeaderBox` in the `MediaInformationBox` as defined in 8.4.5. The volumetric visual media header contains general presentation information, independent of the coding, for volumetric visual media. This header shall be used in any track containing volumetric visual media.

12.10.2.2 Syntax

```
aligned(8) class VolumetricVisualMediaHeaderBox
   extends FullBox('vvhd', 0, 0) {
}
```

12.10.3 Sample entry

12.10.3.1 Definition

Volumetric visual media tracks shall use a `VolumetricVisualSampleEntry`.

12.10.3.2 Syntax

```
class VolumetricVisualSampleEntry(codingname)
   extends SampleEntry (codingname){
   unsigned int(8)[32] compressorname;
   // other boxes from derived specifications
}
```

12.10.3.3 Semantics

`compressorname` is a name, for informative purposes. It is formatted in a fixed 32-byte field, with the first byte set to the number of bytes to be displayed, followed by that number of bytes of displayable data encoded using UTF-8, and then padding to complete 32 bytes total (including the size byte). The field may be set to 0.

12.10.4 Sample format

The format of a volumetric visual sample is defined by the coding system.

12.11 Haptic media

12.11.1 Media handler

Haptic media uses the `'hapt'` handler type in the `HandlerBox` of the `MediaBox`, as defined in 8.4.3.

12.11.2 Media header

Haptics tracks use the `NullMediaHeaderBox` in the `MediaInformationBox` as defined in 8.4.5.

12.11.3 Sample entry

12.11.3.1 Definition

Haptic tracks use `HapticSampleEntry`.

The `HapticSampleEntry` extends the `SampleEntry` class and holds haptic configuration information. This configuration information may be further extended using boxes for codec-specific information.

12.11.3.2 Syntax

```
aligned(8) class HapticSampleEntry(codingname)
   extends SampleEntry(codingname) {
   Box()[]   otherboxes;
}
```

12.11.4 Sample format

The haptics coding format defines the format of a haptics sample. It also defines whether the coding format is all sync-sample, and if not, defines what a sync sample is.

Annex A
(informative)

Background and tutorial

A.1 Annex overview

This annex provides an introduction to the file format, that potentially assists readers in understanding the overall concepts underlying the file format.

A.2 Design considerations

A.2.1 Usage

A.2.1.1 Multi-purpose

The file format is intended to serve as a basis for a number of operations. In these various roles, it may be used in different ways, and different aspects of the overall design exercised.

A.2.1.2 Interchange

When used as an interchange format, the files would normally be self-contained (not referencing media in other files), contain only the media data actually used in the presentation, and not contain any information related to streaming. This will result in a small, protocol-independent, self-contained file, which contains the core media data and the information needed to operate on it.

Figure 1 gives an example of a simple interchange file, containing two streams.

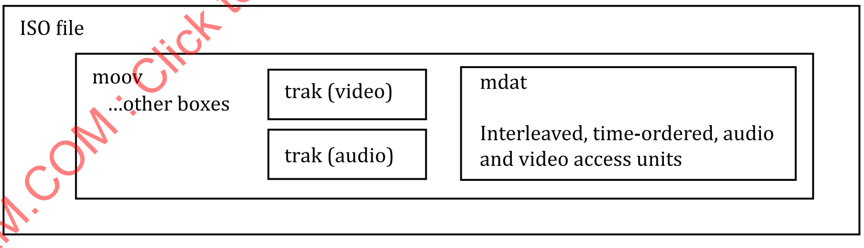

Figure 7 — Simple interchange file

A.2.1.3 Content creation

During content creation, a number of areas of the format can be exercised to useful effect, particularly:

— the ability to store each elementary stream separately (not interleaved), possibly in separate files.

— the ability to work in a single presentation that contains media data and other streams (e.g. editing the audio track in the uncompressed format, to align with an already-prepared video track).

These characteristics mean that presentations may be prepared, edits applied, and content developed and integrated without either iteratively re-writing the presentation on disc – which would be necessary if interleave was required and unused data had to be deleted; and also without iteratively decoding and re-encoding the data – which would be necessary if the data must be stored in an encoded state.

In Figure 2, a set of files being used in the process of content creation is shown.

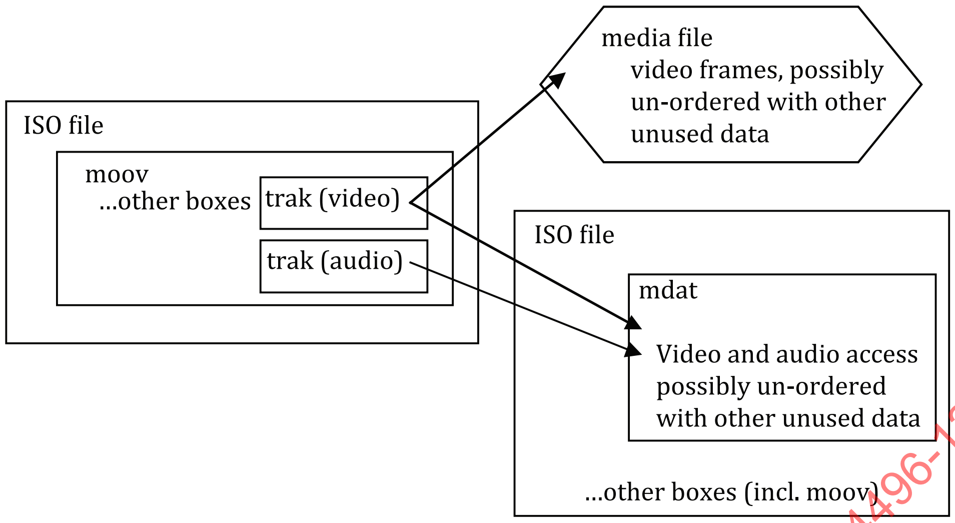
media file
video frames, possibly un-ordered with other unused data
ISO file
moov
...other boxes
trak (video)
trak (audio)
ISO file
mdat
Video and audio access possibly un-ordered with other unused data
...other boxes (incl. moov)

Figure 8 — Content creation file

A.3 Design principles

The file structure is object-oriented; a file can be decomposed into constituent objects very simply, and the structure of the objects inferred directly from their type.

Media-data is not 'framed' by the file format; the file format declarations that give the size, type and position of media data units are not physically contiguous with the media data. This makes it possible to subset the media-data, and to use it in its natural state, without requiring it to be copied to make space for framing. The structure-data is used to describe the media data by reference, not by inclusion.

The file format does not require that a single presentation be in a single file. This enables both sub-setting and re-use of content. When combined with the non-framing approach, it also makes it possible to include media data in files not formatted to this document (e.g. 'raw' files containing only media data and no declarative information, or file formats already in use in the media or computer industries).

The file format is based on a common set of designs and a rich set of possible structures and usages. The same format serves all usages; translation is not required. However, when used in a particular way (e.g. for local presentation), the file may need structuring in certain ways for optimal behaviour (e.g. time-ordering of the data). No normative structuring rules are defined by this document, unless a restricted profile is used.

A.4 Core concepts

In the file format, the overall presentation is called a **movie**. It is logically divided into **track**s; each track represents a timed sequence of media (frames of video, for example). Within each track, each timed unit is called a **sample**; this might be a frame of video or audio. Samples are implicitly numbered in sequence. Note that a frame of audio may decompress into a sequence of audio samples (in the sense this word is used in audio); in general, this document uses the word sample to mean a timed frame or unit of data. Each track has one or more **sample description**s; each sample in the track is tied to a description by reference. The description defines how the sample may be decoded (e.g. it identifies the compression algorithm used).

Unlike many other multi-media file formats, this format, with its ancestors, separates several concepts that are often linked. Understanding this separation is key to understanding the file format. In particular:

The physical structure of the file is not tied to the physical structures of the media itself. For example, many file formats ‘frame’ the media data, putting headers or other data immediately before or after each frame of video; this file format does not do this.

Neither the physical structure of the file, nor the layout of the media, is tied to the time ordering of the media. Frames of video need not be laid down in the file in time order (though they may be).

This means that there are file structures that describe the placement and timing of the media; these file structures permit, but do not require, time-ordered files.

All the data within a conforming file is encapsulated in **box**es (called **atoms** in predecessors of this file format). There is no data outside the box structure. All the structure-data, including that defining the placement and timing of the media, is contained in structured boxes. This document defines the boxes. The media data (frames of video, for example) is referred to by this structure-data. The media data may be in the same file (contained in one or more boxes), or can be in other files; the structure-data permits referring to other files by means of URLs. The placement of the media data within these secondary files is entirely described by the structure-data in the primary file. They need not be formatted to this document, though they may be; it is possible that there are no boxes, for example, in these secondary media files.

Tracks can be of various kinds. Three are important here. **Video tracks** contain samples that are visual; **audio track**s contain audio media. **Hint track**s are rather different; they contain instructions for a streaming server in how to form packets for a streaming protocol, from the media tracks in a file. Hint tracks can be ignored when a file is read for local playback; they are only relevant to streaming.

A.5 Physical structure of the media

The boxes that define the layout of the media data are found in the sample table. These include the data reference, the sample size table, the sample to chunk table, and the chunk offset table. Between them, these tables allow each sample in a track to be both located, and its size to be known.

The **data reference**s permit locating media within secondary media files. This allows a composition to be built from a ‘library’ of media in separate files, without actually copying the media into a single file. This greatly facilitates editing, for example.

The tables are compacted to save space. In addition, it is expected that the interleave will not be sample by sample, but that several samples for a single track will occur together, then a set of samples for another track, and so on. These sets of contiguous samples for one track are called **chunk**s. Each chunk has an offset relative to the containing resource that is identified through a reference to a data reference entry. For example, a chunk offset could be provided relative to the beginning of the file. Within the chunk, the samples are contiguously stored. Therefore, if a chunk contains two samples, the position of the second may be found by adding the size of the first to the offset for the chunk. The chunk offset table provides the offsets; the sample to chunk table provides the mapping from sample number to chunk number.

Note that in between the chunks (but not within them) there may be ‘dead space’, un-referenced by the media data. Thus, during editing, if some media data is not needed, it can simply be left unreferenced; the data need not be copied to remove it. Likewise, if the media data is in a secondary file formatted to a ‘foreign’ file format, headers or other structures imposed by that foreign format can simply be skipped.

A.6 Temporal structure of the media

Timing in the file can be understood by means of a number of structures. The movie, and each track, has a **timescale**. This defines a time axis which has a number of ticks per second. By suitable choice of this number, exact timing can be achieved. Typically, this is the sampling rate of the audio, for an audio track. For video, a suitable scale should be chosen. For example, a media `TimeScale` of 30000 and media sample durations of 1001 exactly define NTSC video (often, but incorrectly, referred to as 29.97) and provide 19.9 hours of time in 32 bits.

The time structure of a track may be affected by an **edit list**. These provide two key capabilities: the movement (and possible re-use) of portions of the timeline of a track, in the overall movie, and also the insertion of 'blank' time, known as empty edits. Note in particular that if a track does not start at the beginning of a presentation, an initial empty edit is needed.

The overall duration of each track is defined in headers; this provides a useful summary of the track. Each sample has a defined **duration**. The exact decoding timestamp of a sample is defined by summing the durations of the preceding samples.

A.7 Interleave

The temporal and physical structures of the file may be aligned. This means that the media data has its physical order within its container in time order, as used. In addition, if the media data for multiple tracks is contained in the same file, this media data would be interleaved. Typically, in order to simplify the reading of the media data for one track, and to keep the tables compact, this interleave is done at a suitable time interval (e.g. 1 second), rather than sample by sample. This keeps the number of chunks down, and thus the chunk offset table small.

A.8 Composition

If multiple audio tracks are contained in the same file, they are implicitly mixed for playback. This mixing is affected by the overall track **volume**, and the left/right **balance**.

Likewise, video tracks are composed, by following their layer number (from back to front), and their composition mode. In addition, each track may be transformed by means of a matrix, and also the overall movie transformed by **matrix**. This permits both simple operations (e.g. pixel doubling, correction of 90º rotation) as well as more complex operations (shearing, arbitrary rotation, for example).

Derived specifications may over-ride this default composition of audio and video with more powerful systems (e.g. MPEG-4 BIFS).

A.9 Random access

This clause describes how to seek. Seeking is accomplished primarily by using the child boxes contained in the `SampleTableBox`. If an edit list is present, it must also be consulted.

To seek a given track to a time T, where T is in the time scale of the `MovieHeaderBox`, the following operations could be performed:

1) If the track contains an edit list, determine which edit contains the time T by iterating over the edits. The start time of the edit in the movie time scale must then be subtracted from the time T to generate T', the duration into the edit in the movie time scale. T' is next converted to the time scale of the track's media to generate T''. Finally, the time in the media scale to use is calculated by adding the media start time of the edit to T''.

2) The `TimeToSampleBox` for a track indicates what times are associated with which sample for that track. Use this box to find the first sample prior to the given time.

3) The sample that was located in step 1 may not be a sync sample. The `SyncSampleBox` documents sync samples, which are random access points (note that there may be random access points that are not documented by the `SyncSampleBox`). Using this table, the first sync sample prior to the specified time can be located. The absence of the sync sample table indicates that all samples are synchronization points, and makes this problem easy. Having consulted the sync sample table, the next step is likely to be seeking to whichever resultant sample is closest to, but prior to, the sample found in step 1.

4) At this point the sample that will be used for random access is known. Use the `SampleToChunkBox` to determine in which chunk this sample is located.

5) Knowing which chunk contained the sample in question, use the `ChunkOffsetBox` to figure out where that chunk begins.

6) Starting from this offset, use the information contained in the `SampleToChunkBox` and the `SampleSizeBox` to figure out where within this chunk the sample in question is located. This is the desired information.

A.10 Fragmented movie files

This clause introduces a technique that may be used in ISO files, where the construction of a single `MovieBox` in a movie is burdensome. This can arise in at least the following cases:

— Recording. If un-fragmented movie files are used, if a recording application crashes, runs out of disk, or some other incident happens, after it has written a lot of media to disk but before it writes the `MovieBox`, the recorded data is unusable. This occurs because the file format insists that all structure-data (the `MovieBox`) be written in one contiguous area of the file.

— Recording. On embedded devices, particularly still cameras, there is not the RAM to buffer a `MovieBox` for the size of the storage available, and re-computing it when the movie is closed is too slow. The same risk of crashing applies, as well.

— HTTP fast-start. If the movie is of reasonable size (in terms of the `MovieBox`, if not time), the `MovieBox` can take an uncomfortable period to download before fast-start happens.

— Segmented streaming. Segment-based streaming systems divide presentations into a sequence of segments which are transmitted and played. An initial segment (often called an initialization segment) contains a `MovieBox` in which the track(s) contain no samples, and subsequent segments contain one or more `MovieFragmentBox`(es).

The basic 'shape' of the movie is set in initial `MovieBox`: the number of tracks, the available sample descriptions, width, height, composition, and so on. However the `MovieBox` does not contain the information for the full duration of the movie; in particular, it may have few or no samples in its tracks.

To this minimal or empty movie, extra samples are added, in structure called movie fragments.

The basic design philosophy is the same as in the `MovieBox`; data is not 'framed'. However, the design is such that it can be treated as a 'framing' design if that is needed. The structures map readily to the `MovieBox`, so a fragmented presentation can be rewritten as a single `MovieBox`.

The approach is that defaults are set for each sample, both globally (once per track) and within each fragment. Only those fragments that have non-default values need include those values. This makes the common case — regular, repeating, structures — compact, without disabling the incremental building of movies that have variations.

The regular `MovieBox` sets up the structure of the movie. It may occur anywhere in the file, though it is best for readers if it precedes the fragments. (This is not a rule, as trivial changes to the `MovieBox` that force it to the end of the file would then be impossible). This `MovieBox`:

— must represent a valid movie in its own right (though the tracks may have no samples at all);

— has a box in it to indicate that fragments should be found and used;

— is used to contain the complete edit list (if any).

Note that software that doesn't understand fragments will play just this initial movie. Software that does understand fragments and gets a non-fragmented movie won't scan for fragments as the fragment indication `MovieExtendsBox` won't be found.

A.11 Construction of fragmented movies

When constructing a fragmented file for playback, there are some recommendations for structuring the content which would optimize playback and random access. The recommendations are as follows:

— The file should consist of boxes in the following order:

  — FileTypeBox

  — MovieBox

  — pair of MovieFragmentBox and MediaDataBox (arbitrary number)

  — MovieFragmentRandomAccessBox

— A MovieFragmentBox consists of at most one TrackFragmentBox for each media. When the file contains a single video track and a single audio track, the MovieFragmentBox will contain two TrackFragmentBoxes, one for the video and one for the audio.

— For video, random accessible samples are stored as the first sample of each TrackFragmentBox. In the case of gradual decoder refresh, a random accessible sample and the corresponding recovery point are stored in the same movie fragment. For audio, samples having the closest presentation time for every video random accessible sample are stored as the first sample of each TrackFragmentBox. Hence, the first samples of each media in the MovieFragmentBox have the approximately equal presentation times.

— First (random accessible) samples are recorded in the MovieFragmentRandomAccessBox for both video and audio.

— All samples in MediaDataBox are interleaved with an appropriate interleave depth.

The offset and the initial presentation time of every MovieFragmentBox are given in the MovieFragmentRandomAccessBox for both audio and video.

The player will load the MovieBox and MovieFragmentRandomAccessBox initially, and hold them in memory during playback. When random access is needed, the player will search MovieFragmentRandomAccessBox in order to find the random access point having the closest presentation time for the indicated time.

Since the first sample in the MovieFragmentBox is random accessible, the player can directory jump in on the random access point. The player can read the MovieFragmentBox of the random access point from the beginning. The subsequent MediaDataBox starts from the random accessible sample. As such, a two-step seeking would not be necessary for random access.

Note that an MovieFragmentRandomAccessBox box is optional, and might never occur in a given file.

A.12 Transformed streaming over streaming protocols

A.12.1 Design considerations for streaming protocols

A.12.1.1 Preparation for transformed streaming

When prepared for transformed streaming, the file needs to contain information to direct the streaming server in the process of sending the information. In addition, it is helpful if these instructions and the media data are interleaved so that excessive seeking can be avoided when serving the presentation. It is also important that the original media data be retained unscathed, so that the files may be verified, or re-edited or otherwise re-used. Finally, it is helpful if a single file can be prepared for more than one protocol, so differing servers may use it over disparate protocols.

A.12.1.2 Local presentation

'Locally' viewing a presentation (i.e. directly from the file, not over a streamed interconnect) is an important application; it is used when a presentation is distributed (e.g. on CD or DVD ROM), during the process of development, and when verifying the content on streaming servers. Such local viewing shall be supported, with full random access. If the presentation is on CD or DVD ROM, interleave is important as seeking may be slow.

A.12.1.3 Streamed presentation

When a server operates from the file to make a stream, the resulting stream has to be conformant with the specifications for the protocol(s) used, and should contain no trace of the file-format information in the file itself. The server needs to be able to random access the presentation. It can be useful to re-use server content (e.g. to make excerpts) by referencing the same media data from multiple presentations; it can also assist streaming if the media data can be on read-only media (e.g. CD) and not copied, merely augmented, when prepared for streaming.

Figure 3 shows a presentation prepared for streaming over a multiplexing protocol, only one hint track is required.

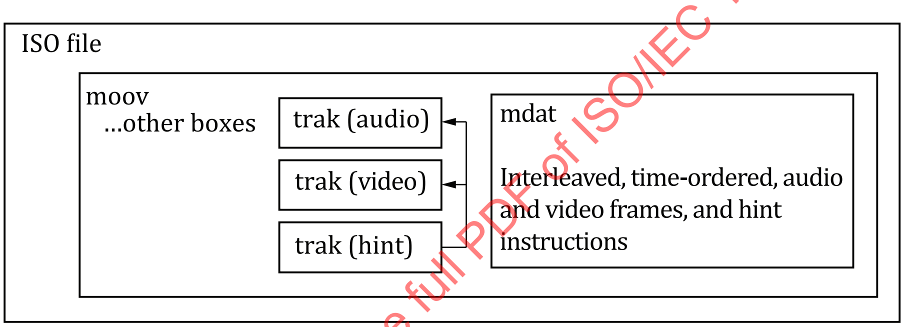
ISO file
moov
...other boxes
trak (audio)
trak (video)
trak (hint)
mdat
Interleaved, time-ordered, audio and video frames, and hint instructions

Figure 9 — Hinted presentation for streaming

A.12.2 Design considerations for streaming protocols

A.12.2.1 General

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 parts 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 protocol information for a particular streaming protocol does not frame the media data; the protocol headers are not physically contiguous with the media data. Instead, the media data can be included by reference. This makes it possible to represent media data in its natural state, not favouring any protocol. It also makes it possible for the same set of media data to serve for local presentation, and for multiple protocols.

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 Players can 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.

A.12.2.2 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.

If an ISO file contains hint tracks, the media tracks that reference the media data from which the hints were built shall remain in the file, even if the data within them is not directly referenced by the hint tracks; after deleting all hint tracks, the entire un-hinted presentation shall remain. The media tracks may, however, refer to external files for their media data.

The protocol information is built in such a way that the streaming servers need to know only about the protocol and the way it should be sent; the protocol information abstracts knowledge of the media so that the servers are, to a large extent, media-type agnostic. Similarly, the media-data, stored as it is in a protocol-unaware fashion, enables the media tools to be protocol-agnostic.

Annex B
(informative)

Guidance on deriving from this document

B.1 General

This annex provides guidance explaining how to derive a specific file format from the ISO base media file format.

This document 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.

B.2 General principles

B.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[22]), and the Motion JPEG 2000 MJ2 file format (ISO/IEC 15444-3[23]). 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 maintenance agency (see Annex D) 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 document, it's worth observing that there are some characteristics that are intentionally ‘parameters’ to the lower specification (this document), that need to be specified. Equally, there are some characteristics of this document that are internal and should rarely be discussed by other specifications. Of course, there are some characteristics in between.

Derived specifications are ideally written solely in terms of the parameters of the file format in this document; what a sample is, what its decoding and composition times 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 `UserDataBox`, or an extension box).

B.2.2 Base layer operations

It should be possible to perform some operations on a file based on this document without knowing anything about any potential derived specifications. These operations might include the obvious reading tracks, finding the data and timing for samples, and their sample description and track type, and so on. This might be done, for example, by a file-format inspector or general library like the reference software.

Less obvious are a class of manipulations of the files:

a) re-interleaving the data; making the media data in time order, with the samples for various tracks grouped into chunks of a sensible size, with the chunks interleaved;

b) making files that use data references self-contained, by copying the data from external files into the new file;

c) removing free space boxes and compacting the box structure;

d) removing data from `MediaDataBox`es that appears to be un-referenced by tracks, items, or structure-data boxes;

e) removing sample entries that have no associated samples;

f) removing sample groups that have no associated samples;

g) extracting some tracks and making a new file with just those (e.g. an audio track from an audio/video presentation);

h) inserting, or removing, movie fragments, or re-fragmenting a movie.

This list is not exhaustive, of course.

B.3 Boxes

Boxes can be added to the file format, but be careful about how they interact with other boxes. In particular, if they 'cross-link' into existing boxes, it might not be possible to mark such files as compliant with this document.

All new boxes must be registered, except those using the `'uuid'` type. Likewise, codec (sample entry) names, brands, track reference types, handlers (media types), group types, and protection scheme types should be registered. It really is a bad idea to use one of these without registration, as collisions may occur – or someone else may register the same identifier with a different meaning.

A box should not be written using the 'UUID escape' (the reserved ISO UUID pattern 0xXXXXXXXX-0011-0010-8000-00AA00389B71, where the four-character code replaces the Xs) if a simple four character code can be used, and ideally it should not be designed to use a UUID box; it's better to place data in known 'expansion points' of the file format if at all possible, or register a new box type if really needed.

Don't forget that *all* data in ISO files must be, or be contained in, boxes. A signature can be introduced, but it must 'look like' a box.

Do not require that any existing or new boxes you define be in a particular position, if at all possible. For example, the existing JPEG 2000 specifications require a signature box and that it be first in the file. If another specification also defines a signature box and also requires that it be first, then a file conformant to both specifications cannot be constructed.

It must be possible to 'walk' the top-level of a file by finding box lengths. Don't forget that 'implied length' is permitted at file level.

Unless absolutely unavoidable, boxes should contain either data (e.g. in fields), or other boxes, but not both. All boxes containing data should be a full box to allow later changes to syntax and semantics. Boxes containing other boxes are known as container boxes, and are normally a plain (non-full) box, since their semantics will never change if they are documented to contain only boxes.

B.4 Brand identifiers

B.4.1 Overview

This subclause covers the use of brand identifiers in the file-type box, including:

— Introduction of a new brand.

— Player's behaviour depending on the brand.

— Setting of the brand on the creation of the ISO base media file.

Brands identify a specification and make a simple set of statements:

a) the file conforms to all requirements of the identified specification;

b) the file contains nothing contrary to the identified specification;

c) a reader implementing potentially that single specification may read, interpret, and possibly present the file, ignoring data it does not recognize.

Specifications should therefore say (if they need a brand) "the brand that identifies files conformant to this specification is XXXX", and register the brand.

B.4.2 Usage of the brand

In order to identify the specifications to which the file complies, brands are used as identifiers in the file format. These brands are set in the `FileTypeBox`.

For example, a brand might indicate:

1) the codecs that may be present in the file,

2) how the data of each codec is stored,

3) constraints and extensions that are applied to the file.

New brands may be registered if it is necessary to make a new specification that is not fully conformant to the existing standards. For example, 3GPP allows using AMR and H.263 in the file format. Since these codecs were not supported in any standards at that time, 3GPP specified the usage of the SampleEntry and template fields in the ISO base media format as well as defining new boxes to which these codecs refer. Considering that the file format is used more widely in the future, it is expected that more brands will be needed.

Brands are not additive; they stand alone. It is not possible to say: "this brand indicates that support for Y is also required" because the 'also' has no referent.

Systems that re-write files should remove brands that they do not recognize, as they do not know whether the file still conforms to that brand's requirements (e.g. re-interleaving a file may take it out of conformance with a specification that requires a certain style of interleaving).

Note that the major brand usually implies the file extension, which in turn implies the MIME type. But these are not rules. In addition, when serving under a MIME type do not forget that MIME types can take parameters, and the list of compatible brands would often be useful to the receiving system.

B.4.3 Introduction of a new brand

A new brand can be defined if conformance to a new specification must be indicated. This generally means that for the definition of a new brand at least one of the following conditions should be satisfied:

1) Use of a codec that is not supported in any existing brands.