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

ISO/IEC JTC1/SC29/WG11
MPEG98/N2612p4
Dec 1998

Source: MPEG-4 Systems
Title: Intermedia Format (MP4) VM text
Authors: David Singer, William Belknap (Editors)
Organization: Apple Computer Inc., IBM Corporation
Group: MPEG-4
Subgroup: Systems

1. Introduction

The MP4 file format is designed to contain the media information of an MPEG-4 presentation in a flexible, extensible format which 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 (a TransMux).

The file format is designed to be independent of any particular TransMux while enabling efficient support for TransMuxes in general.

The design is based on the QuickTime® format from Apple Computer Inc.

1.1 Open Issues

1. Examine the implications of the file format in a live recording and/or sending scenario. [Ming Lee, Mahesh]
2. need some kind of definition, model, mux etc. for local playback (see Appendix A) [Mike Coleman]
3. Need a containment picture [Dave S]
4. Apple to provide a (permanent) pointer to useful docs [Apple]
5. Seeking needs update to cope with edit lists [Hoddie]
6. Need two descriptor tags for use within the file only
7. Need pre-defined 'empty' SLConfigDescriptor

1.2 Goals

The goals for the format are derived from the Call for Proposals, document ISO/IEC JTC1/SC29/WG11/N1919 "Call for Proposals for an MPEG-4 Intermedia format."

1.2.1 TransMux Independence (CFP 3)

MPEG-4 is designed primarily to be delivered as a stream. However, it is recognized that a variety of protocols and streaming substrates will be used to carry that stream. Therefore the file format is designed to support a variety of TransMuxes, without favoring one in particular, either in the design of the format, or in the way the media data is stored in any particular file.

1.2.2 TransMux Support (CFP 12)

However, it is important that the file format support TransMuxes efficiently. The process of delivering a stream over a particular choice of TransMux must be reasonably efficient in both space and time.

1.2.3 Elementary stream management (CFP 4)

MPEG-4 presentations are composed of a number of elementary streams of different types. In the process of forming a presentation (editing), it is necessary that new streams be added or deleted relatively painlessly; preferably without having to re-multiplex the entire presentation before the layup is complete.

The process of forming a presentation may well use formats which start outside the MPEG-4 community: raw audio, for example, is more easily worked during editing than compressed. The ability to handle such ‘mixed’ presentations gracefully is a powerful tool. It is even more powerful if the base media data which is being used in this way can be used in its original container format, even if that is not formatted to this specification. In this way, MPEG-4 files fit into a multimedia community, and their creation and development is facilitated.

1.2.4 Extensibility (CFP 5)

File formats have a long life. They must support extension — room for private fields and ISO defined extensions — in a way which is both non-disruptive yet makes clear what is baseline, and what is extension, so that it is possible to locate and ignore these private fields.

The format should be able to handle extension in objects — new fields and containers for them; elementary stream types; codings for existing elementary streams (e.g. new compression techniques); and TransMuxes.

1.2.5 Exchange format (CFP 6)

Even though delivery of the content is by streaming, it is essential that the files themselves be suitable for the exchange of, and publishing of, material in non-streamed formats (tape, CDROM, DVD, …). This enables a community to grow around the format and develop content collaboratively, and for the content to be used in the broadest possible way.

When used as an exchange format, it is helpful if the file is not burdened with support for streaming or a particular TransMux.

Finally, it can also facilitate use of the content if it may be prepared for streaming, and streamed, over a particular TransMux without requiring the copying or re-writing of the entire presentation. This eases the handling of exchanged content, particularly on read-only media.

1.2.6 Scaling (CFP 7)

There are a number of areas of scalability: protocols may support scalable selection from the media data (e.g. to fit bandwidth constraints); elementary stream codings may also be scalable, or it may be desired to use alternative codings to meet differing needs within one presentation (e.g. fitting to a bandwidth); and the presentations may range over a wide range of sizes, complexity, and bit-rates.

1.2.7 Random access (CFP 8,9)

During streaming, local viewing, and editing, it is important that the system reading the file be able to index, by time, into the presentation, and find how to stream or present the presentation from that time.

1.2.8 Sync, no decode, location, IPR (CFP 1,2, 10, 11)

In order to support editing, and general management of the content, it should not be necessary to 'decode' the format (e.g. by expanding huffman compression codes from some fixed point) in order to read it; this hampers editing, random access, and general management of the presentation. In general, important structures in the file should be accessible by simple steps and not involve searching or decoding.

During presentation, the elementary stream data must be presented and managed in a synchronized fashion: this is basic to any time-based system. In addition, it can be important to carry original or industry-standard timing (e.g. SMPTE time-codes) with a presentation when it is being built.

1.3 Usage

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.

1.3.1 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 over TransMuxes. This results in a small, protocol-independent, self-contained file, which contains the core media data and the information needed to operate on it.

1.3.2 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 which contains MPEG-4 data and other streams (e.g. editing the audio track in uncompressed format, to align with an already-prepared MPEG-4 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 (if interleave has to be maintained, or media data can be left unused), and without iteratively decoding and re-encoding the data (if it must be stored in an encoded state).

1.3.3 Preparation for streaming

When prepared for streaming, the file must 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 it may be used by differing servers over disparate protocols.

1.3.4 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 must be supported, with full random access. If the presentation is on CD or DVD ROM, interleave is important as seeking may be slow.

1.3.5 Streamed presentation

When a server operates from the file to make a stream, the resulting stream must 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.

1.4 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 which give the size, type and position of media data units is 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 meta-data is used to describe the media data by reference, not by inclusion.

Similarly the protocol information for a particular TransMux does not frame the media data; the protocol headers are not physically contiguous with the media data. Instead, the media data is included by reference. This makes it possible to represent media data in its

natural state, not favoring any TransMux. It also makes it possible for the same set of media data to serve for local presentation, and for multiple TransMuxes.

The protocol information is built in such a way that the streaming servers need know about their 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.

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 specification (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), profiles may be used to define the optimal structures and use of options for that usage.

1.5 Design overview

1.5.1 Storage of elementary streams

To maintain the goals of TransMux independence, the media data is stored in its most 'natural' format, and not fragmented. This enables easy local manipulation of the media data. Therefore media-data is stored as access units, a range of contiguous bytes for each access unit.

1.5.2 Handling of elementary streams

The elementary streams in an MPEG-4 presentation are stored in the media tracks as access units (a single access unit is the definition of a 'sample' for an MPEG-4 media stream). This is true for all stream types in this draft, including such 'meta-information' streams as Object Descriptor and the Clock Reference. The consequences of this are, on the positive side, that the file format treats all streams equally; on the negative side, this means that there are 'internal' cross-links between the streams, which means that adding and removing streams from a presentation will involve more than adding or deleting the track and its associated media-data. Not only must the stream be placed in, or removed from, the scene, but the object descriptor stream may need updating.

In a transmitted bit-stream, the access units in the SL Packets are transmitted on byte boundaries. This means that hint tracks will construct SL Packet headers using the information in the media tracks, and the hint tracks will reference the access units from the media track.

The SLConfigDescriptor for the media track uses a pre-defined value which is used only within the file.

1.5.3 Handling of FlexMux

An intermediate, optional, fragmentation and packetization step, called FlexMux, has been defined in the systems specification. Some TransMuxes may carry a FlexMux stream rather than packetized elementary streams. Flexmux may be employed for a variety of purposes, including, but not limited to:

– reducing wasted network bandwidth caused by SL Packet header overhead when the payload is small;
– reducing required server resources when providing many streams, by reducing the number of disk reads or network writes.

The process of building FlexMux PDUs is necessarily aware of the characteristics of the TransMux into which the FlexMux must be placed. It is not therefore possible to design a TransMux-independent handling of FlexMux. Instead, in those TransMuxes where FlexMux is used, the hint tracks for that TransMux will encapsulate and include the formation of FlexMux packets. It is expected that the design of the hint tracks will, in this case, closely reflect the way that FlexMux is used (e.g. a compact table resembling the MuxCode mode may be needed if the interleave offered by that mode is needed).

Note that in some cases, it may not be possible to create a static FlexMux multiplex via a hint track. Notably, if stream selection is dynamic (for example, based on application feedback), or the choice of muxcode modes or other aspects of flexmux is dynamic, and the FlexMux is therefore created dynamically, it is not possible to create a hint track describing the FlexMux; this is a necessary cost of run-time multiplexing. It may be difficult for a server to create such a multiplex dynamically at runtime, but with this cost comes added flexibility. A server that wished to provide such functionality could weigh the costs and benefits, and choose to perform the multiplexing without the aid of hint tracks.

Several MPEG4 structures are intrinsically linked to FlexMux, and therefore must be addressed in the context of a FlexMux-aware hint track. For example, a stream map table must be supplied to the receiving terminal which maps FlexMux channel IDs to elementary stream IDs. Similarly, if the MuxCode mode of FlexMux is used, a MuxCode mode structure for each MuxCode index used must be defined and supplied to the terminal.

These mappings and definitions may change over time, and there is no normative way in MPEG4 to supply these to the terminals; instead, some mechanism, associated with the overall system design or TransMux used, must be employed. The hinter must store the mappings and definitions. Because they are intimately associated with a particular time-segment of a particular hint track, it is recommended that they be placed in the sample description(s) for that hint track. This description would normally be in the form of

a) a table mapping FlexMux channels to elementary stream IDs.

b) a set of MuxCode mode structure definitions.

It is recommended further that the following format, which is derived from the MPEG4 systems specification, be used for the MuxCode mode definitions.

```
aligned(8) class MuxCodeTableEntry {
      int    i, k;
      bit(8) length;
      bit(4) MuxCode;
      bit(4) version;
      bit(8) substructureCount;
      for (i=0; i<substructureCount; i++) {
            bit(5) slotCount;
            bit(3) repetitionCount;
            for (k=0; k<slotCount; k++){
                  bit(8) flexMuxChannel[[i]][[k]];
                  bit(8) numberOfBytes[[i]][[k]];
            }
      }


}
```

Special attention must also be taken when pausing or seeking a stream which is being transported as part of a FlexMux stream. Pausing or seeking any component stream of a FlexMux must necessarily pause or seek all the streams. When seeking, care must be taken with random access points. These may not be aligned in time in the streams which form the FlexMux, which means that any seek operation cannot start them all at a random access point. Indeed, the random access points of the FlexMux itself are necessarily rather poorly defined under such circumstances. It may be necessary for the server to:

a) examine the track references to determine the base media tracks (elementary streams) which are formed into the FlexMux;
b) find the latest time before the desired seek point such that there is a random access point for all the streams between that time and the seek point, by examining each stream separately;
c) transmit the FlexMux stream from that time.

This will ensure that the terminal has received a random access point for all streams at or prior to the desired seek time. However, it may have to discard data for some streams which occurs before the random access points in those streams.

1.5.4 Handling of TransMuxes

The file format supports streaming of media data over a network as well as local playback. The process of sending protocol data units is time-based, just like the display of time-based data, and is therefore suitably described by a time-based format. A file or ‘movie’ which 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 contain instructions for a streaming server which assist in the formation of packets. These instructions may contain immediate data for the server to send (e.g. header information) or reference segments of the media data. These instructions are encoded in the file in the same way that editing or presentation information is encoded in a file for local playback. Instead of editing or presentation information, information is provided which allows a server to packetize the media data in a manner suitable for streaming using a specific network transport or TransMux.

The same media data is used in a file which contains hints, whether it is for local playback, or streaming over a number of different TransMuxes. Separate 'hint' tracks for different TransMuxes may be included within the same file and the media will play over all such TransMuxes 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 TransMuxes. The media data itself need not be recast or reformatted in any way.

This approach to streaming is more space efficient than an approach that requires that the media information be partitioned into the actual data units which 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 TransMuxes 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 to be streamed (e.g., by the application of error-correcting coding techniques, or by encryption).

1.5.4.1 TransMux 'hint' tracks

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

1) The media data is represented as a set of network-independent standard tracks, which may be played, edited, and so on, as normal;
2) There is a common declaration and base structure for server hint tracks; this common format is protocol independent, but contains the declarations of which protocol(s) are described in the server track(s);
3) There is a specific design of the server hint tracks for each TransMux which 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 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 H.261 video and DVI audio, streamed under RTP, results in a packet stream which is fully compliant with the IETF specifications for packing those codings into RTP.

The hint tracks are built and flagged so that when the presentation is viewed directly (not streamed), they may be ignored.

The specific design of the media data (hint samples), and sample descriptions for a particular TransMux is not defined by MPEG4. Instead, the designer of the system using that TransMux, or the body which owns and defines the TransMux, would define these tracks. Clearly there is an advantage in having standard hint track formats for standard TransMuxes, and their development and publication is encouraged.

2. Scope

3. Normative references

4. Additional references

The QuickTime file format specification, May 1996, in HTML:

http://developer.apple.com/techpubs/quicktime/qtdevdocs/REF/refFileFormat96.htm

and in PDF:

http://developer.apple.com/techpubs/quicktime/qtdevdocs/PDF/QTFileFormat.pdf

5. Definitions

5.1 Glossary

Atom..... an object-oriented building block defined by a unique type identifier and length

Chunk ..... A contiguous set of samples for one stream.

Container Atom..... An atom whose sole purpose is to contain and group a set of related atoms.

Hint Track..... A special track which does not contain media data (an elementary stream). Instead it contains instructions for packaging one or more tracks into a TransMux.

Hinter..... A tool that is run on a completed file to add one or more hint tracks to the file and so facilitate streaming.

Movie Atom..... A container atoms whose sub-atoms define the meta-data for a presentation. (‘moov’).

Movie Data Atom..... A container atom which can hold the actual media data for a presentation (‘mdat’).

Sample ..... An access unit for an elementary stream, . . In hint tracks, a sample defines the formation of one or more TransMux packets.

Sample Table..... An index for the timing and physical layout of the samples in a track.

Track ..... A collection of related samples (q.v.) in an MP4 file. For media data, a track corresponds to an elementary stream. For hint tracks, a track corresponds to a TransMux channel.

5.2 Abbreviations

5.3 Symbols

5.4 Syntax

6. File organization

6.1 Presentation structure

6.1.1 File Structure

A presentation may be contained in several files. One file contains the meta-data for the whole presentation, and is formatted to this specification. This file may also contain all the media data, whereupon the presentation is self-contained. The other files, if used, are not required to be formatted to this specification; they are used to contain media data, and may also contain unused media data, or other information. This specification concerns the structure of the presentation file only. The format of the media-data files is constrained by this specification only in that the media-data in the media files must be capable of description by the meta-data defined here.

An MP4 file must contain its media data in media tracks. Hint tracks are optional, though some servers may require hint tracks to be present for a file to be streamed. A file may not contain only hint tracks; the original media tracks which contain the media data from which the hints were built must remain in the file, even if the data within them is not directly referenced by the hint tracks.

6.1.2 Object Structure

The file is structured as a sequence of objects; some of these objects may contain other objects. The sequence of objects in the file must contain exactly one presentation meta-data wrapper (the ‘moov’ wrapper). It is usually at the beginning or end of the file, to permit its easy location. The other objects found at this level may be free space, or media-data wrappers.

The fields in the objects are stored in network byte order (big-endian format).

6.1.3 Meta Data and Media Data

The meta-data is contained within the meta-data wrapper (the Movie atom); the media data is contained either in the same file, within media-data atom(s), or in other files. The media data is composed of access units; the media data objects, or media data files, may contain other un-referenced information.

6.1.4 Track Identifiers

The track identifiers used in an MP4 file are unique within that file; no two tracks may use the same identifier.

Each elementary stream in the file is stored as a media track. The track identifier for these tracks must have a value which fits into two bytes (with zeros in the higher two bytes). The lower two bytes are the elementary stream identifier (ES_ID), in an exact identity mapping.

Hint tracks may use track identifier values in the same range, if this number space is adequate (which it generally is). However, hint track identifiers may also use larger values of track identifier, as their identifiers are not mapped to elementary stream identifiers. Thus very large presentations can use the entire 16-bit number space for elementary stream identifiers.

The next track identifier value in the movie header generally contains a value one greater than the largest track identifier value found in the file. This enables easy generation of a track identifier under most circumstances. However, if this value is equal to or larger than 65535, and a new media track is to be added, then a search must be made in the file for a free track identifier. If the value is all 1s (32-bit maxint), then this search is needed for all additions.

If it is desired to add a track with a known track identifier (elementary stream identifier) then the file must be searched to ensure that there is no conflict. Note that hint tracks can be re-numbered fairly easily; more care should be taken with media tracks as there may be references to their ES_ID (track ID) in other tracks.

Note that if it is desired to have hint tracks have track IDs outside the allowed range for elementary stream tracks, then next track ID will document the next available hint track ID. Since this is larger than 65535, a search will then always be needed to find a valid elementary stream track ID.

If two presentations are merged, then there may be conflict between their track IDs. In that case, one or more tracks will have to be re-numbered. There are two actions to be taken here:

a) Changing the ID of the track itself, which is easy (track ID in the track header).

b) Changing pointers to it.

The pointers occur in two places: in the file format structure itself, and in the data streams. The file format uses track IDs only through track references, which are easily found and modified. However, track IDs become ES_IDs in the MPEG-4 data, and ES_IDs occur in many places. Thus a hint track is fairly readily re-named; the references from a hint track to a media track easily adjusted if that media track is re-named; but the cross-references within MPEG-4 are not. They occur in several of the system streams, not just in the OD stream, and these streams will have to be inspected and updated appropriately.

6.2 Media Data Structure

6.2.1 Elementary Stream Tracks

In the file format, the media data is stored as access units; for each track the entire ES-descriptor is stored as the sample description(s). The stored SLConfigDescriptor uses a pre-defined value which may only be used in the file. The ES descriptor will be re-written by a hinter for transmission into an appropriate form that the TransMux, and at that time the SLConfig established.

Note that the SL Packet header and payload are byte-aligned, so the placement of the header during hinting is possible without bit-shifting, as each SL Packet and its contained access unit will both start on byte boundaries.

Note also that an access unit must be stored as a contiguous set of bytes. This greatly facilitates the fragmentation process used in hint tracks. The file format is able to describe and use media data stored in other files, however this restriction still applies. Therefore if a file is to be used which contains 'pre-fragmented' media data (e.g. a FlexMux stream on disc), the media data will need to be copied to re-form the access units, in order to import the data into this file format.

The ESDescriptor as stored in the file has the following fields and included structures:

```
class ES_Descriptor extends BaseDescriptor : bit(8) tag=ES_DescrTag {
   bit(16)  ES_ID;
   bit(1) streamDependenceFlag;
   bit(1) URL_Flag;
   const bit(1) reserved=1;
   bit(5) streamPriority;
   if (streamDependenceFlag)
     bit(16) dependsOn_ES_ID;
   if (URL_Flag) {
     bit(8) URLlength;
     bit(8) URLstring[URLlength];
   }
   DecoderConfigDescriptor decConfigDescr;
   SLConfigDescriptor      slConfigDescr;
   IPI_DescrPointer        ipiPtr[0 .. 1];
   IP_IdentificationDataSet ipIDS[0 .. 255];
   IPMP_DescriptorPointer  ipmpDescrPtr[0 .. 255];
   LanguageDescriptor    langDescr[0 .. 255];
   QoS_Descriptor        qosDescr[0 .. 1];
   ExtensionDescriptor     extDescr[0 .. 255];
}
```

`ES_ID` – set 0 as stored; when built into a TransMux, the lower 16 bits of the trackID is used;

`streamDependenceFlag` – set 0 as stored; instead, track references of type 'dpnd' are used;

`URLflag` – typically false, as the stream is in the file, not remote

`streamPriority` – set appropriately

`DecoderConfigDescriptor` – this structure duplicates some information available elsewhere (e.g. streamType) and some which may change in editing (the BitRates)

the remaining descriptors are set appropriately, with the exception of SLConfigDescriptor which is discussed above. Note that the QoSDescriptor also may need re-writing for transmission as it contains information about PDU sizes etc.

6.2.1.1 Object Descriptors

The initial object descriptor and object descriptor streams are handled specially within the file format. Object descriptors contain ES descriptors, which in turn contain information which is TransMux specific. In addition, to facilitate editing, the description of a track is stored as an ESDescriptor in the sample description within that track. It must be taken from there, re-written as appropriate, and transmitted as part of the OD stream when the presentation is streamed.

As a consequence, ES descriptors are not stored within the OD track or initial object descriptor. Instead, the initial object descriptor has a descriptor used only in the file, containing solely the track ID of the elementary stream. When used, this descriptor is replaced by an appropriately re-written ESDescriptor from the referenced track. Likewise, OD tracks are linked to ES tracks by track references. Where an ES descriptor would be used within the OD track, another descriptor is used, which again occurs only in the file. It contains the index into the set of mpod track references that this OD track owns. It is replaced by a suitably re-written ESDescriptor by the hinting of this track.

The ES_ID_Inc is used in the initial object descriptor:

```
class ES_ID_Inc extends BaseDescriptor : bit(8) tag=ES_IDIncTag {
  uint(32) Track_ID;   // ID of the track to use
}
```

The ES_ID_Ref is used in the OD stream:

```
class ES_ID_Ref extends BaseDescriptor : bit(8) tag=ES_IDRefTag {
  bit(16)  ref_index;  // track ref. index of the track to use
}
```

Note that a hinter may need to send more OD events than occur in the OD track (for example, if the ES_description changes at a time when there is no event in the OD track). In general, the OD events explicitly authored into the OD track should be sent. The ES descriptor sent in the OD track would be taken from the description of the temporally next sample in the ES track (in decoding time).

6.2.2 Hint Tracks

Hint tracks are used to describe to a server how to serve the elementary stream data in the file over TransMuxes. Each TransMux has its own hint track format. The format of the hints is described by the sample description for the hint track. Most TransMuxes 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 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.

Hint tracks construct TransMuxes 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. Note that these 'pointers' always point at the actual source of the data; if a hint track is built 'on top' of another hint track, then the second hint track will have direct references to the media track(s) used by the first where data from those media tracks is placed in the TransMux.

6.3 Meta-data Structure (Objects)

The following represents the subset of the QuickTime file specification that is required to define an MP4 file. An object in QuickTime terminology is an Atom.

All objects start with a size and type. The size is an unsigned 32-bit integer, giving the size of the object in bytes. The type is an unsigned 32-bit integer, normally interpreted and presented as four printable characters, for ease of identification.

Atoms 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 UUIDs). The standard MPEG-4 atoms all use compact types (32-bit) and most atoms will use the compact (32-bit) size. Typically only the media data atom(s) may need the 64-bit size.

Note that the size is the entire size of the atom, including the size and type header, fields, and all contained atoms. This facilitates general parsing of the file.

```
aligned(8) class Atom (atomtype,
         optional uint(8)[16] extended-type) {
   uint(32) size;
   uint(32) type = atomtype;
   if (size==1) {
      uint(64) largesize;
   }
   if (atomtype='uuid') {
      uint(8)[16] usertype = extended-type;
   }
}
```

The semantics of these two fields are:

`size` is an integer that specifies the number of bytes in this atom. including all its fields and contained atoms; if size is 1 then the actual size is in largesize

`type` identifies the atom type; standard atoms use a compact type, which is normally four printable characters, to permit ease of identification, and is shown so in the atoms below. User extensions use an extended type; in this case, the type field is set to 'uuid'.

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

clip, crgn, matt, kmat, pnot, ctab, load, imap; track reference types tmcd, chap, sync, scpt, ssrc.

Many objects also contain a version number and flags field:

```
aligned(8) class FullAtom(uint(32) atomtype, uint(8) v, uint(24) f)
      extends Atom(atomtype) {
      uint(8)     version = v;
      uint(24)    flags = f;
 }
```

The semantics of these two fields are:

`version` is an integer that specifies the version of this format of the atom.

`flags` is a map of flags

There are a few data types used in a few places in the file:

```
aligned(8) class ufix88 {
      uint(8)     integer-part;
      bit(8)      fractional-part;
}
aligned(8) class fix88 {
      int(8)      integer-part;
      bit(8)      fractional-part;
}
aligned(8) class fix1616 {
      int(16)     integer-part;
      bit(16)     fractional-part;
}
```

The value in each case may be computed by dividing the field, treated as an integer, by 256 (8 bit fraction) or 65536 (16 bit).

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.

Fields shown as reserved in the atom descriptions should be initialized to the given value on atom creation, copied un-inspected when atoms are copied, and ignored on reading.

An overall view of the normal encapsulation structure is provided in the following table.

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

Note that user data objects may be found in moov or trak atoms, and objects using an extended type may be placed in a wide variety of containers, not just the top level.

moov						*	6.3.1	*container for all the meta-data*
	mvhd					*	6.3.3	*movie header, overall declarations*
	iods					*	6.3.4	*object descriptor*

	trak					*	6.3.5	*container for an individual track or stream*
		tkhd				*	6.3.6	*track header, overall information about the track*
		tref					6.3.7	*track reference container*
		edts					6.3.26REF	*edit list container*
			elst				6.3.27REF	*an edit list*
		mdia				*	6.3.8	*container for the media information in a track*
			mdhd			*	6.3.9	*media header, overall information about the media*
			hdlr				6.3.10	*handler, at this level, the media (handler) type*
			minf			*	6.3.11	*media information container*
				vmhd			6.3.12.1	*video media header, overall information (video track only)*
				smhd			6.3.12.2	*sound media header, overall information (sound track only)*
				hmhd			6.3.12.3	*hint media header, overall information (hint track only)*
				<mpeg>			6.3.12.4	*mpeg stream headers*
				dinf		*	6.3.13	*data information atom, container*
					dref	*	6.3.14	*data reference atom, declares source(s) of media in track*
				stbl		*	6.3.15	*sample table atom, container for the time/space map*
					stts	*	6.3.16	*(composition) time-to-sample number map*
					dtts		6.3.17REF	*decoding time-to-sample number map*
					stss		6.3.22REF	*sync (key, I-frame) sample map*
					stsd	*	6.3.18REF	*sample descriptions (codec types, initialization etc.)*
					stsz	*	6.3.19REF	*sample sizes (framing)*
					stsc		6.3.20REF	*sample-to-chunk, partial data-offset information*
					stco		6.3.21REF	*chunk offset, partial data-offset information*
					stsh		6.3.23REF	*shadow sync*
					stdp		6.3.24	*degradation priority*
mdat							6.3.2	*Media data container*
free							6.3.25REF	*free space*
skip							6.3.25REF	*free space*
udta							6.3.28REF	*user-data, copyright etc.*

6.3.1 Movie Atom

Container: File
Mandatory: Yes
Quantity: Exactly one

The meta-data for a presentation is stored in the single MovieAtom which occurs at the top-level of a file. Normally this atom is first or last in the sequence of atoms in a file, though this is not required.

6.3.1.1 Syntax

```
aligned(8) class MovieAtom extends Atom('moov'){
 }
```

6.3.2 Media Data Atom

Container: File
Mandatory: No
Quantity: Any number

If media data is stored in the same file, then this is the container for the media data. In elementary stream tracks, this atom will contain MPEG-4 data as access units. A presentation may contain zero or more media data atoms. The actual media data follows the type field; its structure is documented by the meta-data (see particularly the sample table).

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

Note that there may be any number of these atoms in the file (including zero, if all the media data is in other files). The meta-data refers to media data by its absolute offset within the file (see the chunk offset atom); so mdat headers and free space may easily be skipped, and files without any atom structure may also be referenced and used.

6.3.2.1 Syntax

```
aligned(8) class MediaDataAtom extends Atom('mdat') {
      uint(8) data[];
 }
```

6.3.2.2 Semantics

`data` is the contained media data

6.3.3 Movie Header Atom

Container: Movie Atom ('moov')
Mandatory: Yes
Quantity: Exactly one

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

6.3.3.1 Syntax

```
aligned(8) class MovieHeaderAtom extends FullAtom('mvhd', version, 0) {
   if (version==1) {
      uint(64) creation-time;
      uint(64) modification-time;
      uint(32) timescale;
      uint(64) duration;
```

```
   } else { // version==0
      uint(32) creation-time;
      uint(32) modification-time;
      uint(32) timescale;
      uint(32) duration;
   }
   bit(32)  reserved = 0x00010000;
   bit(16)  reserved = 0x0100;
   bit(16)  reserved = 0;
   uint(32)[2] reserved = 0;
   bit(32)[9]  reserved =
      { 0x00010000, 0, 0, 0, 0x00010000, 0, 0, 0, 0x40000000 };
   bit(32)[6]  reserved = 0;
   uint(32) next-track-ID;
}
```

6.3.3.2 Semantics

`version` is an integer that specifies the version (0 or 1 in this draft)

`creation-time` is an integer which declares the creation time of the presentation (in seconds since midnight, Jan. 1, 1904

`modification-time` is an integer which declares the most recent time the presentation was modified (in seconds since midnight, Jan. 1, 1904

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

`duration` is an integer which declares length of the presentation (in the scale of the timescale). Note that 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.

`next-track-ID` is an integer which indicates a value to use for the track ID of the next track to be added to this presentation. Note that 0 is not a valid track ID value. This must be larger than the largest track-ID in use. If this value is equal to or larger than 65535, and a new media track is to be added, then a search must be made in the file for a free track identifier which will fit into 16 bits. If the value is all 1s (32-bit maxint), then this search is needed for all additions.

6.3.4 Object Descriptor Atom

Container: Movie Atom ('moov')
Mandatory: Yes
Quantity: Exactly one

This object contains the initial MPEG-4 Object Descriptor, as described in ISO/IEC 14496-1 Subclause 8.6.3.

6.3.4.1 Syntax

```
aligned(8) class ObjectDescriptorAtom
      extends FullAtom('iods', version = 0, 0) {
      InitialObjectDescriptor OD;
}
```

6.3.4.2 Semantics

`OD` is an InitialObject Descriptor as defined in ISO/IEC 14496-1.

6.3.5 Track Atom

Container: Movie Atom (‘moov’)
Mandatory: Yes
Quantity: 1 or more

This is a container atom for a single track of a presentation. A presentation may consist of one or more tracks. Each track is independent of the other tracks in the presentation and carries its own temporal and spatial information. Each track will contain its associated media atom.

Tracks are used for two purposes: (a) to contain elementary media data (media tracks) and (b) to contain packetization information for streaming protocols (hint tracks).

There must be at least one media track within an MP4 file; and all the media tracks which contributed to the hint tracks present, must 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 must remain.

6.3.5.1 Syntax

```
aligned(8) class TrackAtom extends Atom(‘trak’) {
 }
```

6.3.6 Track Header Atom

Container: Track Atom (‘trak’)
Mandatory: Yes
Quantity: Exactly one

The track header atom specifies that characteristics of a single track. Exactly one track header atom is contained in a track.

In the absence of an edit list, the presentation of a track starts immediately. An empty edit is used to offset the start time of a track.

6.3.6.1 Syntax

```
aligned(8) class TrackHeaderAtom
      extends FullAtom('tkhd', version, flags){
      if (version==1) {
            uint(64)    creation-time;
            uint(64)    modification-time;
            uint(32)    track-ID;
            uint(32)    reserved = 0;
            uint(64)    duration;
      } else { // version==0
            uint(32)    creation-time;
            uint(32)    modification-time;
            uint(32)    track-ID;
            uint(32)    reserved = 0;
            uint(32)    duration;
      }
      uint(32)[3] reserved = 0;
      bit(16)     reserved = { if track_is_audio 0x0100 else 0};
      uint(16)    reserved = 0;
      bit(32)[9]  reserved =
            { 0x00010000, 0, 0, 0, 0x00010000, 0, 0, 0, 0x40000000 };
      bit(32)     reserved = {
            if track_is_visual 0x01400000 else 0 };
      bit(32)     reserved = {
            if track_is_visual 0x00F00000 else 0};
}
```

6.3.6.2 Semantics

`version` is an integer that specifies the version (0 or 1 in this draft)

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

Track enabled- Indicates that the track is enabled. Flag value is 0x000001. A disabled track (the low bit is zero) is treated as if it were not present.

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

`modification-time` is an integer which declares the most recent time the track was modified (in seconds since midnight, Jan. 1, 1904)

`track-ID` is an integer which uniquely identifies this track over the entire life-time of this presentation. Track Ids are never re-used and cannot be zero.

`duration` is an integer that indicates the duration of this track (in the movie's time coordinate system). Note that this property is derived from the track's edits. The value of this field is equal to the sum of the durations of all of the track's edits. If there is no edit list, then the duration is the sum of the sample durations, converted into the movie time-scale.

6.3.7 Track reference atom

Container: Track Atom ('trak')
Mandatory: No
Quantity: 0 or 1

The track reference atom provides a reference from the containing stream to another stream in the presentation. These references are typed. In particular, a 'hint' reference

links from the containing hint track to the media data which it hints. Exactly one track reference atom can be contained within the track atom.

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

6.3.7.1 Syntax

```
aligned(8) class TrackReferenceAtom extends Atom('tref') {
 }
aligned(8) class TrackReferenceTypeAtom extends Atom(reference-type) {
      uint(32) track-IDs[];
 }
```

6.3.7.2 Semantics

The track reference atom contains track reference type atoms. These are structured as track reference type atoms.

The `reference-type` must be set to one of the following values:

- `hint` the referenced track(s) contain the original media for this hint track
- `dpnd` the referencing track has an MPEG-4 dependency on the referenced track
- `mpod` the referencing track is an OD track which uses the referenced track as an included elementary stream track

6.3.8 Media atom

Container: Track Atom ('trak')
Mandatory: Yes
Quantity: Exactly one

The media declaration container contains all the objects which declare information about the media data within a stream.

6.3.8.1 Syntax

```
aligned(8) class MediaAtom extends Atom('mdia') {
 }
```

6.3.9 Media header atom

Container: Media Atom ('mdia')
Mandatory: Yes
Quantity: Exactly one

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

6.3.9.1 Syntax

```
aligned(8) class MediaHeaderAtom extends FullAtom('mdhd', version, 0) {
      if (version==1) {
            uint(64)    creation-time;
            uint(64)    modification-time;
            uint(32)    timescale;
            uint(64)    duration;
      } else { // version==0
            uint(32)    creation-time;
            uint(32)    modification-time;
            uint(32)    timescale;
            uint(32)    duration;
      }
      bit(1)      pad = 0;
      uint(5)[3]  language;   // packed ISO-639-2/T language code
      uint(16)    reserved = 0;
 }
```

6.3.9.2 Semantics

`version` is an integer that specifies the version

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

`modification-time` is an integer which declares the most recent time the presentation was modified (in seconds since midnight, Jan. 1, 1904)

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

`duration` is an integer which declares length of this media (in the scale of the timescale).

`language` declares the language code for this media. See ISO 639-2/T for the set of three character codes. 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.

6.3.10 Handler reference atom

Container: Media Atom ('mdia') Atom
Mandatory: Yes
Quantity: 1 only

The handler atom within a Media Atom declares the process by which the media-data in the stream may be presented, and thus, the nature of the media in a stream. For example, a video track would be handled by a video handler.

Note that in this version of the draft, video and audio are not in video or audio tracks, but in MPEG-4 visualstream and audiostream tracks.

6.3.10.1 Syntax

```
aligned(8) class HandlerAtom extends FullAtom('hdlr', version, 0) {
      uint(32)    reserved = 0;
      uint(32)    handler-type;
      uint(8)[12] reserved = 0;
      string      name;
 }
```

6.3.10.2 Semantics

`version` is an integer that specifies the version

`handler-type` is an integer containing one of the following values:

- `'odsm'` ObjectDescriptorStream
- `'crsm'` ClockReferenceStream
- `'sdsm'` SceneDescriptionStream
- `'vide'` VisualStream
- `'soun'` AudioStream
- `'m7sm'` MPEG7Stream
- `'ocsm'` ObjectContentInfoStream
- `'ipsm'` IPMP Stream
- `'mjsm'` MPEG-J Stream
- `'hint'` Hint track

`name` is a null-terminated string in UTF-8 characters which gives a human-readable name for the stream type (for debugging and inspection purposes).

6.3.11 Media information atom

Container: Media Atom ('mdia')
Mandatory: Yes
Quantity: Exactly one

The media information wrapper contains all the objects which declare characteristic information of the media in the stream.

6.3.11.1 Syntax

```
aligned(8) class MediaInformationAtom extends Atom('minf') {
 }
```

6.3.12 Media information header atoms

Container: Media Information Atom ('minf')
Mandatory: exactly one media header must be present
Quantity: 1 only

There is a media information header for each track type (corresponding to the media handler type). This header is used for all tracks containing visual streams.

6.3.12.1 Video Media Header Atom

The video media header contains general presentation information, independent of the coding, for visual media.

6.3.12.1.1 Syntax

```
aligned(8) class VideoMediaHeaderAtom
      extends FullAtom(‘vmhd’, version = 0, 1) {
      uint(64)    reserved = 0;
 }
```

6.3.12.1.2 Semantics

`version` is an integer that specifies the version

6.3.12.2 Sound Media Header Atom

The sound media header contains general presentation information, independent of the coding, for audio media. This header is used for all tracks containing audio streams.

6.3.12.2.1 Syntax

```
aligned(8) class SoundMediaHeaderAtom
      extends FullAtom(‘smhd’, version = 0, 0) {
      uint(32)    reserved = 0;
 }
```

6.3.12.2.2 Semantics

`version` is an integer that specifies the version

6.3.12.3 Hint Media Header Atom

The hint media header contains general information, independent of the protocol, for hint tracks.

6.3.12.3.1 Syntax

```
aligned(8) class HintMediaHeaderAtom
      extends FullAtom(‘hmhd’, version = 0, 0) {
      uint(16)    maxPDUsize;
      uint(16)    avgPDUsize;
      uint(32)    maxbitrate;
      uint(32)    avgbitrate;
      uint(32)    slidingavgbitrate;
 }
```

6.3.12.3.2 Semantics

`version` is an integer that specifies the version

`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

`slidingavgbitrate` gives the maximum rate in bits/second over any window of one minute (corresponding to the avgBitrate field in the DecoderConfigDescriptor)

6.3.12.4 MPEG-4 Media Header Atoms

MPEG-4 streams other than visual and audio currently use an empty media header. There are a set of reserved types for media headers specific to these MPEG-4 stream types.

6.3.12.4.1 Syntax

```
aligned(8) class Mpeg4MediaHeaderAtom
      extends FullAtom('nmhd', version = 0, flags) {
 }
```

6.3.12.4.2 Semantics

`version` is an integer that specifies the version

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

The following types are reserved but currently unused.

ObjectDescriptorStream	`'odhd'`
ClockReferenceStream	`'crhd'`
SceneDescriptionStream	`'sdhd'`
MPEG7Stream	`'m7hd'`
ObjectContentInfoStream	`'ochd'`
IPMP Stream	`'iphd'`
MPEG-J Stream	`'mjhd'`

6.3.13 Data information atom

Container: Media Information Atom ('minf')
Mandatory: Yes
Quantity: Exactly one

The data reference wrapper contains objects which declare the location of the media information in a stream.

6.3.13.1 Syntax

```
aligned(8) class DataInformationAtom extends Atom('dinf') {
 }
```

6.3.14 Data reference atom

Container: Data Information Atom ('dinf')
Mandatory: Yes
Quantity: Exactly one

The data reference object contains a table of data references (normally URLs) which 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 samples. 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 atom, then no string (not even an empty one) should be supplied in the entry field.

6.3.14.1 Syntax

```
aligned(8) class DataEntryUrlAtom(version, flags)
      extends FullAtom('url ', version = 0, flags) {
      string      location;
}
```

```
aligned(8) class DataEntryUrnAtom(version, flags)
      extends FullAtom(‘urn ’, version = 0, flags) {
      string      name;
      string      location;
}

aligned(8) class DataReferenceAtom
      extends FullAtom(‘dref’, version = 0, 0) {
      uint(32)    entry-count;
      for (int i=0; i < entry-count; i++) {
            DataEntryAtom(entry-version, entry-flags) data-entry;
      }
}
```

6.3.14.2 Semantics

`version` is an integer that specifies the version

`entry-count` is an integer which counts the actual entries

`entry-version` is an integer that specifies the version

`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 movie atom

`data-entry` is a URL or URN entry. 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. Each is a null-terminated string usingUTF-8 characters. If the self-contained flag is set; the URL form is used and no string is present; the atom terminates with the entry-flags field. The URL type should be of a service which delivers a file (e.g. URLs of type file, http, ftp etc.), which ideally also permits random access. Relative URLs are permissible and are relative to the file containing this data reference.

6.3.15 Sample Table atom

Container: Media Information Atom (‘minf’)
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 sample table atom references no data, then the sample table atom does not need to contain any sub-atoms (this is not a very useful media track).

If the track that the sample table atom is contained in does reference data, then the following sub-atoms are required: Sample Description, Sample Size, Sample to Chunk, and Chunk Offset. Further, the Sample Description Atom must contain at least one entry. A Sample Description Atom is required because the it contains the data reference index field which indicates which Data Reference atom to use to retrieve the media samples. Without the Sample Description, it is not possible to determine where the media samples are stored. The Sync Sample atom is optional. If the Sync Sample atom is not present, all samples are sync samples.

6.3.15.1 Syntax

```
aligned(8) class SampleTableAtom extends Atom(‘stbl’) {
 }
```

6.3.16 Composition Time to Sample atom

Container: Sample Table Atom (‘stbl’)
Mandatory: Yes
Quantity: Exactly one

This atom contains a compact version of a table which allows indexing from composition time 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 duration, and the duration of those samples. By adding the durations a complete time-to-sample map may be built.

If only composition time stamp is used within the track, then samples are ordered by those time stamps; the durations are therefore all positive. If the track uses decoding time-stamps, then samples are ordered by decoding time. In this case, the durations in the composition time stamp table may be negative, reflecting a re-ordering of the samples for display (composition) purposes.

The CTS axis has a zero origin; CTS(i) = SUM(for j=0 to i-1 of duration(j)), and the sum of all durations gives the length of the media in the track (not mapped to the overall timescale, and not considering any edit list).

6.3.16.1 Syntax

```
aligned(8) class TimeToSampleAtom
      extends FullAtom(’stts’, version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    sample-count;
            int(32)     sample-duration;
      }
}
```

6.3.16.2 Semantics

`version` is an integer that specifies the version
`ttype` is ‘stts’ (for composition times)
`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 which have the given duration
`sample-duration` is an integer that gives the duration of these samples in the time-scale of the media

6.3.17 Decoding Time to Sample atom

Container: Sample Table Atom (‘stbl’)
Mandatory: No
Quantity: Exactly one

This atom provides the offset between decoding time and composition time. Since decoding time must preceed composition time, the offsets are expressed as unsigned numbers such that DTS = CTS – offset.

The decoding time stamp table is optional and should only be present if DTS and CTS differ. In the following example, there is a sequence of frames I1, B2, B3, P4, each with a duration of 10. The samples are stored as follows, with the indicated values for their composition durations and decoding offsets (the actual CTS and DTS are given for reference). The re-ordering occurs because the predicted frame P4 must be decoded before the bi-directionally predicted frames B2 and B3. The value of CTS for a sample is always the sum of the durations of the preceding samples. Note that the total of the composition durations is the duration of the media in this track (40 in this example).

	I1	P4	B2	B3
CTS	0	30	10	20
DTS	-10	0	10	20
Composition duration	30	-20	10	20
Decoding offset	10	30	0	0

6.3.17.1 Syntax

```
aligned(8) class DecodingOffsetAtom
      extends FullAtom('dtts', version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    sample-count;
            uint(32)    decoding-offset;
      }
}
```

6.3.17.2 Semantics

`version` is an integer that specifies the version, 0 in this draft

`ttype` is 'dtts' (for decoding times)

`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 which have the given offset

`sample-offset` is an integer that gives the offset between CTS and DTS, such that DTS(i) = CTS(i) – offset(i)

6.3.18 Sample description atom

Container: Sample Table Atom ('stbl')
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 information stored in the data array is stream-type specific, and may have variants within a stream type (e.g. different codings may use different specific information after some common fields, even within a visual stream).

For visual streams, a VisualSampleEntry is used; for audio streams, an AudioSampleEntry. For all other MPEG-4 streams, an MpegSampleEntry is used. Hint tracks use an entry format specific to their protocol, with an appropriate name.

For all the MPEG-4 streams, the data field stores an ES_Descriptor with all its contents. Note that this provides an SlConfigDescriptor which uses a pre-defined value solely for use within files.

For hint tracks, the sample description contains appropriate declarative data for the TransMux being used, and the format of the hint track. The definition of the sample description is specific to the TransMux. However, note the discussion of FlexMux above, and the need for a Stream Map table, and MuxCode mode format definitions.

Note that multiple descriptions may be used within a stream.

6.3.18.1 Syntax

```
aligned(8) class HintSampleEntry(format) extends Atom(protocol) {
      uint(8)[6]  reserved = 0;
      uint(16)    data-reference-index;
      uint(8)     data[];           // format specific to the protocol
}

aligned(8) class ESDAtom
      extends FullAtom(‘esds’, version = 0, 0) {
      ES_Descriptor     ES;
}
      // Visual Streams
aligned(8) class VisualSampleEntry(format) extends Atom(‘mp4v’) {
      uint(8)[6]  reserved = 0;
      uint(16)    data-reference-index;
      uint(32)[4] reserved = 0;
      uint(32)    reserved = 0x014000F0;
      uint(32)    reserved = 0x00480000;
      uint(32)    reserved = 0x00480000;
      uint(32)    reserved = 0;
      uint(16)    reserved = 1;
      uint(8)     name-len;         // length of following coding name
      char(31)    name;             //  typically “MPEG-4 Visual”
      uint(16)    reserved = 24;
      int(16)     reserved = -1;
      ESDAtom     ES;
}
```

```
      // Audio Streams
aligned(8) class AudioSampleEntry(format) extends Atom('mp4a') {
      uint(8)[6]  reserved = 0;
      uint(16)    data-reference-index;
      uint(32)[2] reserved = 0;
      uint(16)    reserved = 2;
      uint(16)    reserved = 16;
      uint(32)    reserved = 0;
      uint(16)    time-scale; // copied from the track
      uint(16)    reserved = 0;
      ESDAtom     ES;
}

      // all other Mpeg stream types
aligned(8) class MpegSampleEntry(format) extends Atom('esds') {
      uint(8)[6]  reserved = 0;
      uint(16)    data-reference-index;
      ESDAtom     ES;
}

aligned(8) class SampleDescriptionAtom
      extends FullAtom('stsd', version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            SampleEntry(entry-format)     entry;
      }
}
```

6.3.18.2 Semantics

`version` is an integer that specifies the version

`entry-count` is an integer that gives the number of entries in the following table

`SampleEntry` is the appropriate sampleentry.

`data-reference-index` is integer that contains the index of the data reference to use to retrieve data associated with samples that use this sample description. Data references are stored in data reference atoms.

`data` is information specific to the protocol.

`ES` is the ES Descriptor for this stream.

6.3.19 Sample size atom

Container: Sample Table Atom ('stbl')
Mandatory: Yes
Quantity: Exactly one

The sample size atom contains the sample count and a table giving the size of each sample. This allows the media data itself to be unframed.The total number of samples in the track is always indicated in the sample count. If the default size is indicated, then no table follows.

6.3.19.1 Syntax

```
aligned(8) class SampleSizeAtom extends FullAtom(‘stsz’, version = 0, 0)
{
      uint(32)    sample-size;
      uint(32)    sample-count;
      if (sample-size==0) {
            for (int i=0; i < sample-count; i++) {
                  uint(32)    entry-size;
            }
      }
 }
```

6.3.19.2 Semantics

`version` is an integer that specifies the version

`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.

`entry-count` is an integer that gives the number of entries in the following table

`entry-size` is integer specifying the size of a sample, indexed by its number.

6.3.20 Sample to chunk atom

Container: Sample Table Atom (‘stbl’)
Mandatory: Yes
Quantity: Exactly one

Samples within the media data are grouped into chunks. Chunks may be of different sizes, and the samples within a chunk may have different sizes. By using this table, you can 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, you can compute how many chunks are in this run. You can convert this to a sample count by multiplying by the appropriate samples-per-chunk.

6.3.20.1 Syntax

```
aligned(8) class SampleToChunkAtom
      extends FullAtom(‘stsc’, version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    first-chunk;
            uint(32)    samples-per-chunk;
            uint(32)    sample-description-index;
      }
 }
```

6.3.20.2 Semantics

`version` is an integer that specifies the version

`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 which share the same samples-per-chunk and sample-description-index

`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 description which describes the samples in this chunk

6.3.21 Chunk offset atom

Container: Sample Table Atom (‘stbl’)
Mandatory: Yes
Quantity: Exactly one

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.

Note that offsets are file offsets, not the offset into any atom within the file (e.g. an mdat atom). This permits referring to media data in files without any atom structure. It does also mean that care must be taken when constructing a self-contained mp4 file with its meta-data (movie atom) at the front, as the size of the movie atom will affect the chunk offsets to the media data.

6.3.21.1 Syntax

```
aligned(8) class ChunkOffsetAtom
      extends FullAtom(‘stco’, version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    chunk-offset;
      }
 }

aligned(8) class ChunkLargeOffsetAtom
      extends FullAtom(‘co64’, version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(64)    chunk-offset;
      }
 }
```

6.3.21.2 Semantics

`version` is an integer that specifies the version

`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 into its containing media stream (file).

6.3.22 Sync Sample Atom

Container: Sample Table Atom (‘stbl’)
Mandatory: No
Quantity: Exactly one

The sync sample atom provides a compact marking of the random access points within the stream. Precisely the samples named here would have the RandomAccessPoint flag set in their SL Packet headers. The table is arranged in strictly increasing order of sample number.

If this table is not present, every sample is a random access point.

6.3.22.1 Syntax

```
aligned(8) class SyncSampleAtom
      extends FullAtom('stss', version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    sample-number;
      }
 }
```

6.3.22.2 Semantics

`version` is an integer that specifies the version

`entry-count` is an integer that gives the number of entries in the following table

`sample-number` gives the numbers of the samples which are random access points in the stream

6.3.23 Shadow Sync Sample Atom

Container: Sample Table Atom ('stbl')
Mandatory: No
Quantity: Exactly one

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

Each entry in the shadowSyncTable 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 random access at, or before, the shadowed-sample-number.

The entries in the ShadowSyncAtom must be sorted based on the shadowed-sample-number field.

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

The shadowsync sample replaces, not augments, the sample which 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 time-stamps etc. needed in their headers.

6.3.23.1 Syntax

```
aligned(8) class ShadowSyncSampleAtom
      extends FullAtom('stsh', version = 0, 0) {
      uint(32)    entry-count;
            for (int i=0; i < entry-count; i++) {
            uint(32)    shadowed-sample-number;
            uint(32)    sync-sample-number;
      }
 }
```

6.3.23.2 Semantics

`version` is an integer that specifies the version

`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

6.3.24 Degradation Priority Atom ('stdp')

Container: Sample Table Atom ('stbl').
Mandatory: No.
Quantity: Exactly one.

The degradation priority atom contains the MPEG-4 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 Sample Size Atom ('stsz').

The maximum size of a degradation priority in the SL header is 15 bits, A fixed 15-bit field is used here.

6.3.24.1 Syntax

```
aligned(8) class DegradationPriorityAtom
      extends FullAtom('stdp', version = 0, 0) {
      for (int i=0; i < sample-count; i++) {
            bit(1)      pad; // must be zero
            uint(15)    priority;
      }
}
```

6.3.24.2 Semantics

`version` is an integer that specifies the version.

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

6.3.25 Free space wrapper

Container: File
Mandatory: No
Quantity: Any number

The contents of a free-space wrapper are irrelevant and may be ignored, or the object deleted, without affecting the presentation. (Note that deleting the object may invalidate the offsets used in the sample table, unless this object is after all the media data).

6.3.25.1 Syntax

```
aligned(8) class FreeSpaceAtom extends Atom(free-type) {
      uint(8) data[];
 }
```

6.3.25.2 Semantics

`free-type` may be `'free'` or `'skip'`.

6.3.26 Edit Atom

Container: Track Atom ('trak')
Mandatory: No
Quantity: Exactly one

An edit atom maps the presentation time-line to the media time-line as it is stored in the file. The edit atom is a container for the edit lists.

Note that the Edit atom is optional. In the absence of this atom, there is an implicit one-to-one mapping of these time-lines.

In the absence of an edit list, the presentation of a track starts immediately. An empty edit is used to offset the start time of a track.

6.3.26.1 Syntax

```
aligned(8) class EditAtom extends Atom('edts') {
 }
```

6.3.27 Edit List Atom

Container: Edit Atom ('edts')
Mandatory: No
Quantity: 1 only

The edit list atom contains an explicit timeline map. It is possible to represent 'empty' parts if the timeline, where no media is presented; a 'dwell', where a single time-point in the media is held for a period; and a normal mapping.

Starting offsets for tracks (streams) are represented by an initial empty edit. For example, 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

Segment-duration = 10 seconds
Media-Time = -1
Media-Rate = 1.0

Segment-duration = 30 seconds (could be the length of the whole track)
Media-Time = 0 seconds
Media-Rate = 1.0

6.3.27.1 Syntax

```
aligned(8) class EditListAtom extends FullAtom('elst', version = 0, 0) {
      uint(32)    entry-count;
      for (int i=0; i < entry-count; i++) {
            if (version==1) {
                  uint(64) segment-duration;
                  int(64) media-time;
            } else { // version==0
                  uint(32)    segment-duration;
                  int(32)     media-time;
            }
            fix1616     media-rate;
      }
 }
```

6.3.27.2 Semantics

`version` is an integer that specifies the version

`entry-count` is an integer that gives the number of entries in the following table

`track-duration` is an integer that specifies the duration of this edit segment in units of the movie's time scale

`media-time` is an integer containing the starting time within the media of this edit segment (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 should never be an empty edit. Any difference between the movie's duration and the track's duration is expressed as an implicit empty edit.

`media-rate` is a fixed-point number that specifies the relative rate at which to play the media corresponding to this edit segment. If this value is 0, then the edit is specifying a 'dwell': the media at media-time is presented for the segment-duration. Otherwise this field must contain the value 1.0.

6.3.28 User-data atom

Container: Movie Atom ('moov') or Track Atom ('trak')
Mandatory: No
Quantity: Any quantity

The stream user-data wrapper contains objects which declare user information about the containing wrapper and its data (presentation or stream).

The user-data atom is a container atom for informative user-data. This user data is formatted as a set of atoms with more specific atom types, which declare more precisely their content.

Only a copyright notice is defined in this draft. There may be multiple copyright atoms using different language codes.

6.3.28.1 Syntax

```
aligned(8) class UserDataAtom extends Atom('udta') {
 }
aligned(8) class CopyrightAtom
      extends FullAtom('cprt', version = 0, 0) {
      bit(1)      pad = 0;
      uint(5)[3]  language;   // packed ISO-639-2/T language code
      string      notice;
 }
```

6.3.28.2 Semantics

`language` declares the language code for the following text. See ISO 639-2/T for the set of three character codes. 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` is a null-terminated string giving a copyright notice

7. Extensibility

7.1 Objects

The normative objects defined in this specification are identified by a 32-bit value, which is normally a set of four printable characters from the ISO 8859-1 character set.

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

Commonly used in distributed computing are UUIDs (universal unique identifiers), which are 16 bytes. Any MPEG-4 normative type specified here may be mapped directly into the UUID space by composing the four byte type value with the twelve byte MPEG reserved value, tentatively 0xxxxxxxxx-0011-0010-8000-00AA00389B71. The four character code replaces the XXXXXXXX in the preceding number. These types are identified to MPEG as the object types used in this specification.

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

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

```
size := read_uint32();
type := read_uint32();
if (type==‘uuid’)
      then uuid := read_uuid()
      else uuid := form_uuid(type, MPEG_12_bytes);
```

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

```
write_uint32( object_size(object) );
uuid := object_uuid_type(object);
if (is_MPEG_uuid(uuid) )
      write_uint32( MPEG_type_of(uuid) )
      else { write_uint32(‘uuid’); write_uuid(uuid); }
```

A file containing MPEG-4 objects which have been written using the ‘uuid’ escape and the full UUID is not compliant; systems may choose to read objects using the uuid escape and an MPEG-4 uuid as equivalent to the MPEG-4 object of the same type as equivalent, or not.

7.2 Elementary streams

MPEG-4 streams may be combined into a presentation with other streams. Such streams and their declarations are beyond the scope of this specification.

7.3 TransMuxes (protocols)

Hint tracks may be defined for a number of protocols. There are informative/normative appendices in this document for the hint tracks for RTP and MPEG-2 Transport.

7.4 Storage formats

The main file containing the meta-data may use other files to contain media-data. These other files may contain header declarations from a variety of standards, including this one.

If such a secondary file has a meta-data declaration set in it, that meta-data is not part of the overall presentation. This allows small presentation files to be aggregated into a larger overall presentation by building new meta-data and referencing the media-data, rather than copying it.

The references into these other files need not use all the data in those files; in this way, a subset of the media-data may be used, or unwanted headers ignored.

8. Appendix A: Local Playback

A set of hint tracks which provide timing and sequencing for local playback, conforming to the systems decoder model for local playback, may be needed. Their existence, format and usage are TBD.

9. Appendix B: TransMux: Real-Time Protocol (IETF)

This section presents a track format for streaming RTP.

Note: this section is tentative, and depends on the resolution of the packing of MPEG-4 into RTP.

RTP is currently being worked on at the IETF; it is currently defined to be able to carry a limited set of media types (principally audio and video) and codings. The packing of MPEG-4 elementary streams into RTP is under discussion in both bodies. The use of RTP as an example protocol in this proposal should not be taken to mean that only RTP is supported, or that the file format favors RTP.

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. Each hint track is tied to a set of media tracks by track references. The set of references form a table, which is indexed by the samples (see below) when selecting data from the media tracks.

This design decides the packet size at the time the 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. Other protocols can be parameterized in a similar way. Similarly the time-scale for the RTP clock is provided.

The sample description for RTP declares the maximum packet size which this hint track will generate. Session description (SAP/SDP) information is stored in user-data atoms in the track.

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

```
aligned(8) class RtpSampleEntry extends SampleEntry('rtp ') {
      uint(32) timescale;
}
```

Each sample in the RTP hint track contains the instructions to send out a set of packets which must be emitted at a given time. The time in the hint track is emission time, not necessarily the media time of the associated media. This is normally the RTP time of the packet.

Notice that we now describe the internal structure of samples, which are media data, not meta data. These need not be structured as objects.

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). Note that the size of the sample is known from the sample size table.

```
aligned(8) class RTPsample {
      uint(16)    packetcount;
      uint(16)    reserved;
      RTPpacket   packets[packetcount];
      byte        extradata[];
}
```

Each RTP packet contains the information to send a single packet. In order to separate media time from emission time, an RTP time stamp is specifically included, along with data needed to form the RTP header. Other header information is supplied; the algorithms for forming the RTP header given the information here are simple. Then there is a table of construction entries:

```
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)      reserved;
      bit(1)      P-bit;
      bit(1)      X-bit;
      bit(4)      reserved;
      bit(1)      M-bit;
      bit(7)      payload-type;

      uint(16)    RTPsequenceseed;
      uint(13)    flags;
      uint(1)     x-flag;
      uint(1)     b-flag;
      uint(1)     r-flag;
      uint(16)    entrycount;
      dataentry   constructors[entrycount];
}
```

The relative-time field ‘warps’ the actual transmission time away from the sample time. This allows traffic smoothing. The x-flag indicates that there is extra information after the constructors, in the form of type-length-value sets. Only one such set is currently defined; ‘rtpo’ gives a 32-bit signed integer offset to the actual RTP time-stamp to place in the packet. This enables packets to be placed in the hint track in decoding order, but have their presentation time-stamp in the transmitted packet be in a different order.

The b-flag indicates a disposable ‘b-frame’. The r-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.

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) {
      uint(8)     constructor-type = type;
}
```

```
aligned(8) class RTPnoopconstructor
      extends RTPconstructor(0)
{
}

aligned(8) class RTPimmediateconstructor
      extends RTPconstructor(1)
{
      uint(8)     count;
      uint(8)     data[count];
      uint(8)     pad[15-count];
}

aligned(8) class RTPsampleconstructor
      extends RTPconstructor(2)
{
      uint(8)     trackrefindex;
      uint(16)    length;
      uint(32)    samplenumber;
      uint(32)    sampleoffset;
      uint(16)    bytesperblock;
      uint(16)    samplesperblock;
}

aligned(8) class RTPsampledescriptionconstructor
      extends RTPconstructor(3)
{
      uint(8)     trackrefindex;
      uint(16)    length;
      uint(32)    sampledescriptionindex;
      uint(32)    sampleoffset;
}
```

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 bytesperblock and samplesperblock concern compressed audio. This allows translation of the samplenumber into an actual byte offset in the audio track.

The sampledescription mode allows sending of sample descriptions (which would contain decoder config descriptors) as part of an RTP packet.

For complex cases (e.g. encryption or forward error correction), the transformed data would be placed into the hint samples, and then hintsample mode would be used. Note that this would be from the extradata field in the RTPsample itself.

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).

10. Appendix C: TransMux: MPEG-2 Transport

This section presents a simple track format for streaming MPEG-2 transport from a file holding elementary streams.

An MPEG-2 transport stream describes a multiplex of one or more elementary streams. For this reason, an MPEG-2 transport hint track describes how to construct such a multiplex from one or more media tracks. There is not a one to one relationship between media tracks and MPEG-2 transport hint tracks. Each hint track contains references to the elementary streams it represents. A file might contain multiple such hint tracks to describe different multiplexes.

Packet size is not an issue, since all MPEG-2 transport packets are 188 bytes in size. Each transport packet only contains payload data from one media track. This allows for a simple hint description for each transport packet. Each such hint must describe which header data appears on each transport packet, and then point to the payload in the appropriate media track for the transport packet. For packets which do not correspond with a media track, such as PSI packets, the hint will describe 188 bytes of header data, and any media track reference will be irrelevant. For packets which do correspond with a media track, the header data will account for things such as transport headers, possible adaptation headers, and PES headers for transport packets that begin PES packets.

The MPEG-2 transport hint tracks contain the usual declarations for hint tracks. To allow for the very large sizes which MPEG movies can be, we declare in the sample description of the hint track, the field sizes used in the samples for offset and size information (four byte or eight byte field sizes). It is also required that the hint track be associated with the media tracks which it streams. This is done through the use of standard track references. To keep the samples compact, the samples indicate the base media track, by using its index in the track reference table associated with this hint track.

We must also describe the format of the hint samples themselves. Note that these are media data, not meta-data. Consequently, the samples need not be structured as objects.

Each sample describes one transport packet. Each transport packet can be described as some amount of header data, followed by some amount of payload from one media track. Since MPEG-2 transport packets are relatively small, we will have a large number of samples, and care must be taken to keep these samples as small as possible. Several entries in the additional data table are used to minimize the size of samples as much as possible, but these will make some of the fields in the sample entries variable in size. When this is the case, it is described below.

```
struct mpeg2sample {
      int(8)      mediatrackreference;
      int(8)      headerdatalength;
      byte[headerdatalength]  headerdata;
            -- the sizes of these 2 fields are declared in the track
      int()       mediasamplenumber;
      int()       mediasampleoffset;
}
```

Note that for these samples it is not necessary to indicate the length of the payload data for the packet. It is always equal to 188 minus the size of the header data for the packet.

The header data length is always less than, or equal to, 188. If it is 188 then the media track reference, sample number, and sample offset are irrelevant and can be ignored.

11. Appendix D: Example file

12. Appendix E: Layout of Media Data

The MP4 format provides a great deal of flexibility in how the media data is physically arranged within a file. This flexibility is desirable and useful. However, it can allows media layouts to be created which may be inefficient for play back on a given device. It should also be noted, however, that a media layout which is inefficient for a given device may be very efficient for another. Therefore, it is not the intention of this discussion to define a given type of media layout as wrong. Rather, the intention is to define some common uses of MP4 files and describe the media layout in these circumstances.

An MP4 file can reference media data stored in a number of files, including the MP4 file itself. If an MP4 file only references media data contained within itself, the MP4 file is said to be "self-contained".

An MP4 file can reference media data stored in files that are not MP4 files. This is because the MP4 format references media within a URL by file offset, rather than by a data structuring mechanism of a particular file format. This allows an MP4 file to refer to data stored in any container format.

It is often convenient to store a single elementary stream per file, for example when encoding content. It is also useful for content reuse — to reuse an elementary stream, it is not necessary to extract it from a larger, possibly multiplexed file. Because MP4 can reference media stored in any file, it is not required that elementary stream files be stored in the MP4 format. However, this is recommended. Putting the MP4 wrapper around an elementary stream has several advantages, particularly in enabling interchange of the content between different tools (since MP4 is the only normative file format for MPEG-4 media). Further, the MP4 format adds very little overhead to the elementary stream — as little as a few hundred bytes in many cases — so there is no great penalty in storage space. It may be useful to give a name to these types of files — MP4 files that contain a single elementary stream — so that they can be consistently described.

One of the issues facing any device (server, local workstation) that is attempting to play back an MP4 file in real time is the number of file seeks that must be performed. It is possible to arrange the data in an MP4 file to minimize, and potentially eliminate, any seeks during the course of normal playback. (Of course random access and other kinds of interactivity will require seeks.) This is accomplished by interleaving the elementary stream data in the MP4 file in such a way that the layout of the media in the file corresponds to the order in which the media data will be required. It is expected that most servers, for example, will stream MPEG-4 media using the facilities of the hint track. Let's take a scenario where the MP4 file contains a single hint track which reference an audio and a visual elementary stream. In order to eliminate all seeks, the hint track media must be interleaved with the audio and visual stream data. Furthermore, because the hint track sample must always be read before the audio and/or visual media that it references, the

hint track samples must always immediately precede the samples they reference. Here's a simple illustration of the ordering of data (time and file offset increasing from left to right).

H0 A0 H1 V1 H2 V2 H3 A1 H4 A2 V3 H5 V4

Note that when a single hint sample references to multiple pieces of elementary stream data, those pieces of media data must be occur in the order that they are referenced.

13. Appendix F: Random Access

Seeking within an mp4 file is accomplished using the subatoms contained in the Sample Table atom. When asked to seek a given track to a time T, you may perform the following operations on the sample table atoms:

1) The time-to-sample atom for a track indicates what times are associated with which sample for that track. Use this track to find the first sample prior to the given time.
2) The sample that was located in step one may not be a random access point. Locating the nearest random access point requires consulting two atoms. The sync sample table indicates which samples are in fact random access points. Using this table, you can locate which is the first sync sample prior to the specified time. The absence of the sync sample table indicates that all samples are synchronization points, and makes this problem easy. The shadow sync atom gives the opportunity for a content author to provide samples that are not delivered in the normal course of delivery, but which can be inserted to provide additional random access points. This improves random access without impacting bitrate during normal delivery. This atom maps samples that are not random access points to alternate samples which are. You should also consult this table if present to find the first shadow sync sample prior to the sample in question. Having consulted the sync sample table and the shadow sync table, you probably wish to seek to whichever resultant sample is closest to, but prior to, the sample found in step 1.
3) At this point you know the sample that will be used for random access. Use the sample-to-chunk table to determine in which chunk this sample is located.
4) Knowing which chunk contained the sample in question, use the chunk offset atom to figure out where that chunk begins.
5) Starting from this offset, you can use the information contained in the sample-to-chunk atom and the sample size atom to figure out where within this chunk the sample in question is located. This is the desired information.