Matroska File Format (under construction!)

Contents

1 Introduction

This document is intended to be used by developers who want to implement support for the *Matroska* file format in their applications, but who want to build this support from scratch rather than using existing implementations, or people who just want to understand the *Matroska* file format in detail. Thus, the file format itself is described, but not the usage of any existing library including the official one ('libmatroska').

This document does not replace the documentation on
`http://www.matroska.org/technical/specs/index.html`
Especially, default values of elements in case they are not written to a file should be looked up on that page, as they are not listed again in this document.

2 EBML - basics

2.1 Unsigned Integer values of variable length ("vlint")

Files based on EBML use integers of variable length for element IDs and to store the size of elements. The length of an integer is equivalent to

```
length = 1 + [number of leading zero bits]
```

All integers use big endian.

Example: 3A 41 FE:

The first byte 3A (**00**11 1010) has 2 leading zeros, resulting in a total length of 3. The final value of this integer is obtained by changing the most significant bit being 1 to 0. In case of 3A 41 FE, the bit to swap is 00**1**1 1010. The result is then 0x1A41FE. Another possibility to write 1A41FE would be `10 1A 41 FE` or `08 00 1A 41 FE`. When writing EBML files, the shortest possible encoding should be used to avoid wasting space.

Unknown Length

All bits after the leading zeros being set to one, such as FF or 7F FF, indicates an *unknown length*. Muxers shall avoid writing unknown length values whenever possible. The only exception is the last Level 0 element of a file. If encoding a number as described above results in such a sequence, it shall be encoded again with a greater destination length. Example: When encoding 8191 as described above, the result is 7F FF. In 7F FF, all bits after the leading zero are set, which would indicate an unknown length. That means, the length is increased to 3, and the number is encoded again to 20 3F FF.

Note

It is possible to use a lookup table to determine the total length from the first byte. The Matroska file format does not allow integer lengths greater than 8, meaning that the number of leading zeros is not higher than 7 and that the total length can always be retrieved from the first byte.

2.2 Signed Integer values of variable length (svlint)

Signed integers have the following value: Read the integer as Unsigned Integer and then subtract

```
vsint_subtr[length-1]
```

where

```
__int64 vsint_subtr [] =
    { 0x3F, 0x1FFF, 0x0FFFFF, 0x07FFFFFF,
      0x03FFFFFFFF, 0x01FFFFFFFFFF,
      0x00FFFFFFFFFFFFFF, 0x007FFFFFFFFFFFFF };
```

2.3 Signed and Unsigned Integers (int and uint)

Integers, signed as well as unsigned, are stored 'normally' and in big endian byte order, with leading 0x00 and 0xFF being cut off (example for int: -257 is 0xFE 0xFF).

2.4 Float

A `Float` value is a 32 or 64 bit real number, as defined in IEEE. 80 Bit values have been in the specification, but have been removed and should not be used. The bytes are stored in big endian order.

2.5 Types of Strings

String refers to an ASCII string.
UTF-8 refers to a string that is encoded as UTF-8

2.6 Usage of EBML elements

One piece of information is stored the following way:

```
typedef struct {
    vlint      ID        // EBML-ID
    vlint      size      // size of element
    char[size] data      // data
} EBML_ELEMENT;
```

The length of ID shall be called `s_ID`, the length of size shall be called `s_size`. Elements that contain other EBML Elements are called *EBML Master elements*.

Generally, the order of EBML elements inside a parent element is not fixed. In some cases, a certain order is recommended, but it is never mandatory. Especially, no element order should be assumed inside small parent elements.

3 Level 0 Elements in *Matroska* files

3.1 EBML

This header describes the contents of an EBML file. There should be only one EBML header in one file. Any further EBML headers do not render a file invalid, but shall be ignored by any application reading the file.

Files with more than one EBML header could be created for instance if two or more files are appended by using the `copy /b` command (which is not recommended, but no one can be prevented from doing this)

3.2 Segment

A `Segment` contains multimedia data, as well as any header data necessary for replay. There can be several `Segment`s in one *Matroska* file, but this is not encouraged to be done, as there are no players available which are capable of replaying multisegment *Matroska* files correctly.

4 Level 1 - Element: EBML - Header

4.1 EBMLVersion (int)

`EBMLVersion` indicates the version of the EBML Writer that has been used to create a file.

4.2 EBMLReadVersion (int)

`EBMLReadVersion` indicates the minimum version an EBML parser needs to be compliant with to be able to read the file.

4.3 EBMLMaxIDLength (int)

`EBMLMaxIDLength` indicates the length of the longest EBML-ID the file contains. In case of matroska, this value is 4. Any EBML-ID which is longer than the value of this element shall be considered invalid.

4.4 EBMLMaxSizeLength (int)

`EBMLMaxSizeLength` indicates the maximum `s_size` value the file contains. Any EBML element having an `s_size` value greater than EBMLMaxSize-Length shall be considered invalid.

4.5 DocType (string)

`DocType` describes the contents of the file. In the case of a *Matroska* file, its value is `'matroska'`.

4.6 DocTypeVersion (int)

`DocTypeVersion` indicates the version of the `<DocType>` writer used to create the file.

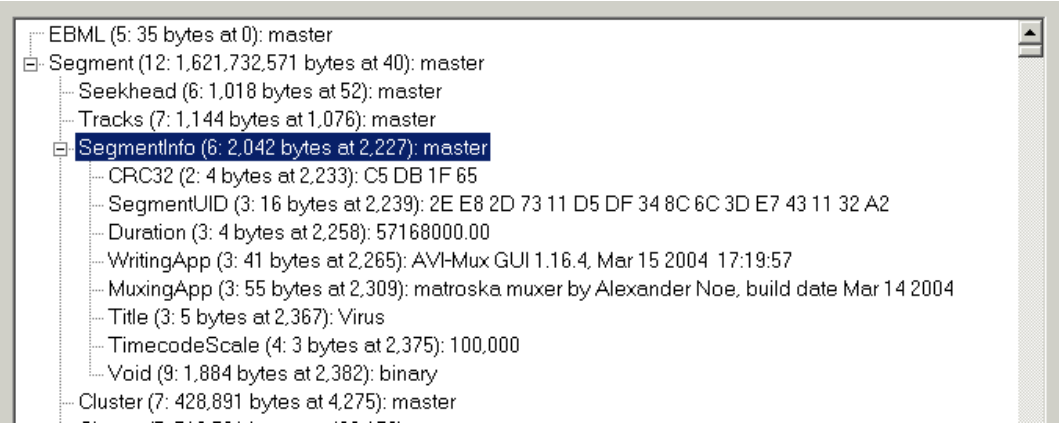
4.7 DocTypeReadVersion (int)

`DocTypeReadVersion` indicates the minimum version number a `<DocType>` parser must be compliant with to read the file.

5 Level 1 - Elements inside Segments

5.1 SegmentInfo

The `SegmentInfo` element contains general information about the `Segment`, such as its duration, the application used for writing the file, date of creation, a unique 128 bit ID, to name a few only. Information included in the `SegmentInfo` element is not required for playback, but should be written by any *Matroska* muxer.



(read: <element name> (<s_size + s_ID>: <size> bytes at <position in file>: value)

5.1.1 SegmentUID (char[16])

The `SegmentUID` is a unique 128 bit number identifying a `Segment`. It shall be written by any *Matroska* muxer, but shall not be considered mandatory by a parser.

5.1.2 SegmentFilename (utf-8)

`SegmentFilename` contains the name of the file the `Segment` is stored in. Since renaming files is easy, the value of this element shall not be considered reliable.

5.1.3 PrevUID (char[16])

`PrevUID` contains the unique 128 bit ID of the `Segment` that is replayed before the currently active `Segment`, i.e. the ID of the `Segment` that should

be loaded if seeking to a timecode earlier than the earliest timecode of active `Segment` is tried. That `segment` should, of course, be easy to locate, for instance in a file in the same directory.

5.1.4 PrevFilename (utf-8)

`PrevFilename` contains the name of the file in which the `Segment` having the ID `<PrevUID>` is stored. `PrevFilename` shall not be considered reliable for the same reason as `SegmentFilename`.

5.1.5 NextUID (char[16])

`NextUID` contains the unique 128 bit ID of the `Segment` that is replayed after the currently active `Segment`, i.e. the ID of the `Segment` that should be loaded if seeking to a timecode after the end of the active `Segment` is tried. Like `PrevUID`, the corresponding `Segment` should be easy to locate.

5.1.6 NextFilename (utf-8)

`NextFilename` contains the name of the file in which the `Segment` having the ID `NextUID` is stored. `NextFilename` shall not be considered reliable for the same reason as `SegmentFilename`.

5.1.7 TimecodeScale (int)

Each scaled timecode in a *Matroska* file is multiplied by this value to obtain a timecode in nanoseconds. Note that not all timecodes are scaled!

5.1.8 Duration (float)

The `Duration` element indicates the duration of the `Segment`. The duration measured in nanoseconds is equal to `<Duration> * <TimecodeScale>`. This element should be written.

5.1.9 DateUTC

?

5.1.10 Title

Contains a general name of the `Segment`, like `''Lord of the Rings - The Two Towers‘‘`. This element is not mandatory.

5.1.11 MuxingApp

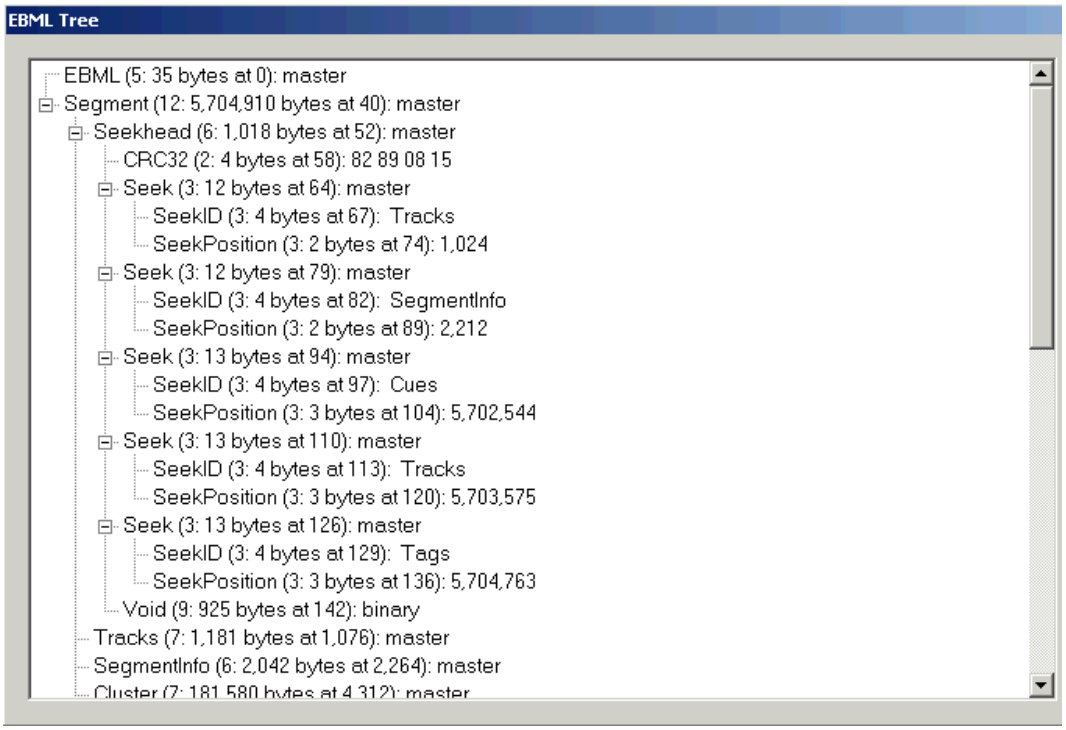
Contains the name of the library that has been used to create the file (like "libmatroska 0.6.0“). This element should be written by any muxer! Especially if non-compliant files are encountered, this element will help to find out who is to be blamed.

5.1.12 WritingApp

Cotains the name of the application used to create the file (like "mkvmerge 0.8.1“). This element should be written for the same reason as `MuxingApp`.

5.2 Seekhead

The `Seekhead` element contains a list of positions of Level 1 elements in the `Segment`. Each pair (element id, position) is stored in one `Seek` element:

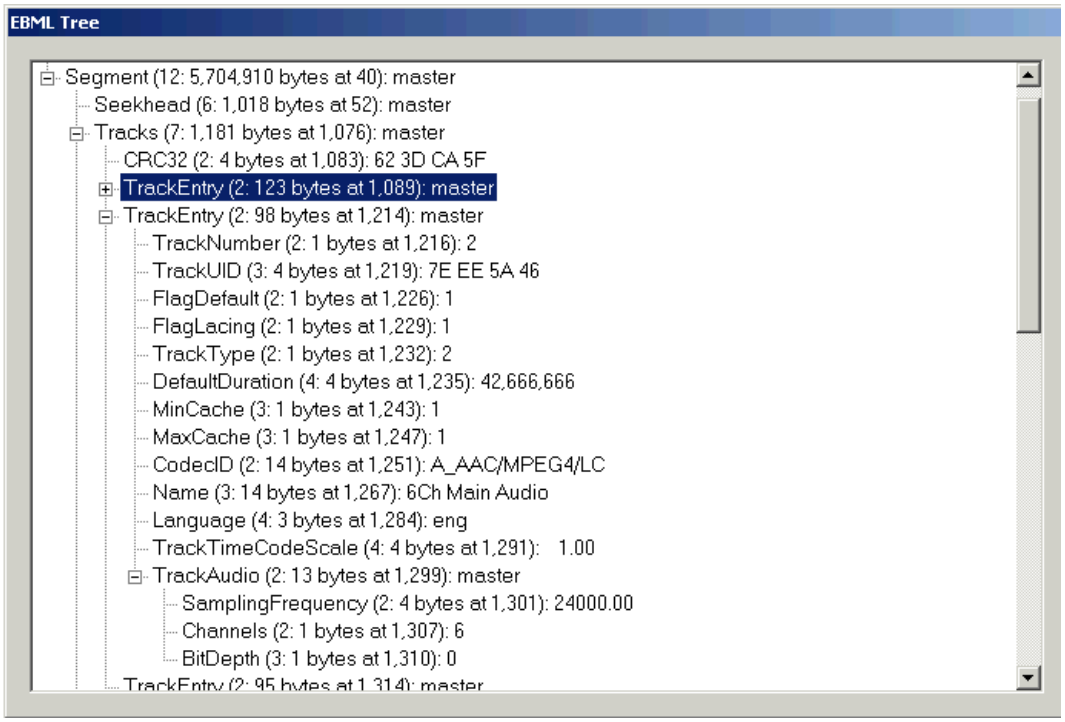


`SeekID` contains the EBML-ID, and `SeekPosition` the position inside the client area of the `Segment` of the corresponding element.

Not all Level 1 elements need to be included. Each `Segment` should contain a `Seekhead` element. Typical `Seekhead`s either include a list of all Level 1 elements, or a list of all Level 1 elements except for `Clusters` (see section 5.4). `Seekhead`s can also include references to other `Seekhead`s. Circles shall never occur in such cases.

5.3 Tracks

The `Tracks` element contains information about the tracks that are stored in the `Segment`, like track type (audio, video, subtitles), the used codec, resolution and sample rate. All tracks shall be described in one (or more, but preferably only one) `Tracks` element. The contents of a `Segment` cannot be processed without the `Tracks` element. Not finding any `Tracks` element inside a `Segment` shall be considered a fatal error.



As you can see, there are several `TrackEntries` in the `Tracks` element. Each `TrackEntry` describes one track of the file. In theory, several `TrackEntries` could refer to the same track, but that is not recommended, except for backup purposes!

5.3.1 TrackNumber (int)

Defines an identification number of the track. This number shall not be equal to zero.

5.3.2 TrackUID (int32)

This 32-bit integer is a unique identificator of the track within the file.

5.3.3 TrackType (int)

Defines the type of a track and can be

- `Video: 0x01`
- `Audio: 0x02`
- `Complex: 0x03 (e.g. DV with combined Video+Audio)`
- `Logo: 0x10`
- `Subtitle: 0x11`
- `Control: 0x20`

5.3.4 FlagDefault (int)

Select whether or not the track should be enabled by default.

5.3.5 FlagLacing (int)

This flag indicates whether or not Lacing (see page 26) is used in the corresponding track. The reliability of this flag should not be overrated.

5.3.6 MinCache (int)

Indicates the number of frames a player must be able to cache during playback. This is for instance interesting if a native MPEG4 file with frames in coding order is played.

5.3.7 MaxCache (int)

Indicates the maximum number of frames a player has to cache. A value of NULL means that no cache is required.

5.3.8 DefaultDuration (int)

This value indicates the number of nanoseconds a frame lasts. This value is applied if no Duration value is indicated for a frame or if Lacing (see page 26) is used. A value of 0 means that the duration of frames of the track is not constant (e.g. variable framerate video, or Vorbis audio).

5.3.9 TrackTimecodeScale (float)

Every timecode of a block (`cluster timecode + block timecode`) is multiplied by this value to obtain the real timecode of a block.

5.3.10 TrackOffset (int)

This value is added to every timecode of blocks of the corresponding track. It can be used to adjust sync between video and audio without remuxing the entire file.

5.3.11 Name (UTF-8)

A human-readable name for the track

5.3.12 Language (string)

Specifies the language of a track, using ISO-639-2 (see http://lcweb.loc.gov/standards/iso639-2/englangn.html)

5.3.13 CodecID (string)

Specifies the Codec which is used to decode the track. A complete list of currently supported CodecID strings can be found on http://matroska.org/technical/specs/codecid/index.html .

5.3.14 CodecPrivate (binary)

Contains information the Codec needs before decoding can be started. An example is the Vorbis initialization packets for Vorbis audio.

5.3.15 CodecName (UTF-8)

A human-readable name of the Codec

5.3.16 CodecSettings (UTF-8)

A string describing the settings used for encoding.

5.3.17 CodecInfoURL (String)

Contains an URL where to find information about that codec

5.3.18 CodecDownloadURL (String)

Indicates an URL where the specified codec can be downloaded.

5.3.19 CodecDecodeAll (uint)

If this value is set to nonzero, the codec can handle potentially damaged data. That means that damaged data should be passed over to the codec rather than being dropped, e.g. if reading data from a damaged Mode 2 Form 2 - CD.

5.3.20 TrackOverlay (uint)

This element contains the number of the track (TrackNumber) which it is an overlay of.

5.3.21 video

Contains video-specific information about the track. This element is only present for video tracks.

- **FlagInterlaced (int)**
  If this flag is set to nonzero, the track contains interlaced video.
- **StereoMode (int)**
  0: no stereo, 1: right eye, 2: left eye, 3: both eyes
- **PixelWidth (int)**
  Width of the encoded video track in pixels
- **PixelHeight (int)**
  Height of the encoded video track in pixels

- **DisplayWidth (int)**
  Width the video track shall be zoomed to when replaying
- **DisplayHeight (int)**
  Height the video track shall be zoomed to when replaying
- **DisplayUnit (int)**
  Indicates the unit DisplayWidth and DisplayHeight are indicated in.
  0: Pixels, 1: centimeters, 2: inches
- **AspectRatioType (int)**
  Indicates the default behaviour of players when resizing the replay window:
  0: free resizing, 1: keep aspect ratio, 2: fixed
- **ColorSpace (string)**
  Same value as in AVI
- **GammaValue (float)**
  Gamma value

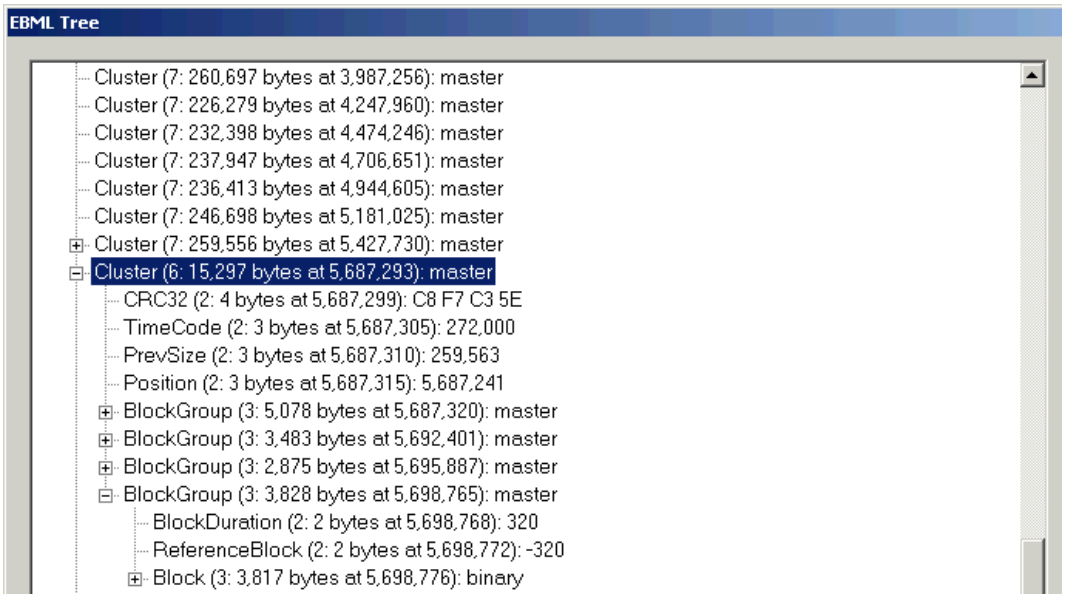
5.3.22 audio

Contains audio-specific information about a track. This element is only present for audio tracks.

- **SamplingFrequency (float)**
  Indicates the sample rate the track is encoded at
- **OutputSamplingFrequency (float)**
  Indicates the sample rate the track should be output at. This value is often equal to `SamplingFrequency`. It differs for instance for HE-AAC audio, which uses SBR.
- **Channels**
  Indicates the number of channels
- **BitDepth**
  Indicates the number of bits per sample of the track. This value is usually used for PCM audio and may be zero for other formats, even if the correct value is known.

5.4 Cluster

A `Cluster` contains multimedia data and usually spans over a range of a few seconds. The layout of a `Cluster` is the following:



Although sticking to this order of the elements is not mandatory, it is recommended not to have any non-`BlockGroup` after the first `BlockGroup`.

5.4.1 Timecode (int)

The Cluster timecode is the timecode all block timecodes are indicated relatively to.

5.4.2 Position (int)

The `Position` element indicates the position of the beginning of its parent element inside its grand parent element. This can help to resync in case of damaged data, but is of no use if no data is damaged.

5.4.3 PrevClusterSize (int)

Indicates the size of the preceding cluster in bytes. This helps to seek backwards, and to find the preceding cluster, without having to look at `Metaseek`

or `Cue` data. This is also helpful to resync, e.g. if the EBML-ID of the preceding `Cluster` is damaged.

5.4.4 BlockGroup

A Blockgroup contains a single block, as well as some additional information concerning that block.

- **ReferenceBlock (sint)**
  Contains the timecode (relatively to the current block) of a `block` that needs to be decoded before the current one can be processed. In case of bidirectionally encoded frames, there can be more than one `ReferenceBlock`.

- **BlockDuration (int)**
  Indicates the scaled duration of the `block`. If this value is not written, it is assumed to be

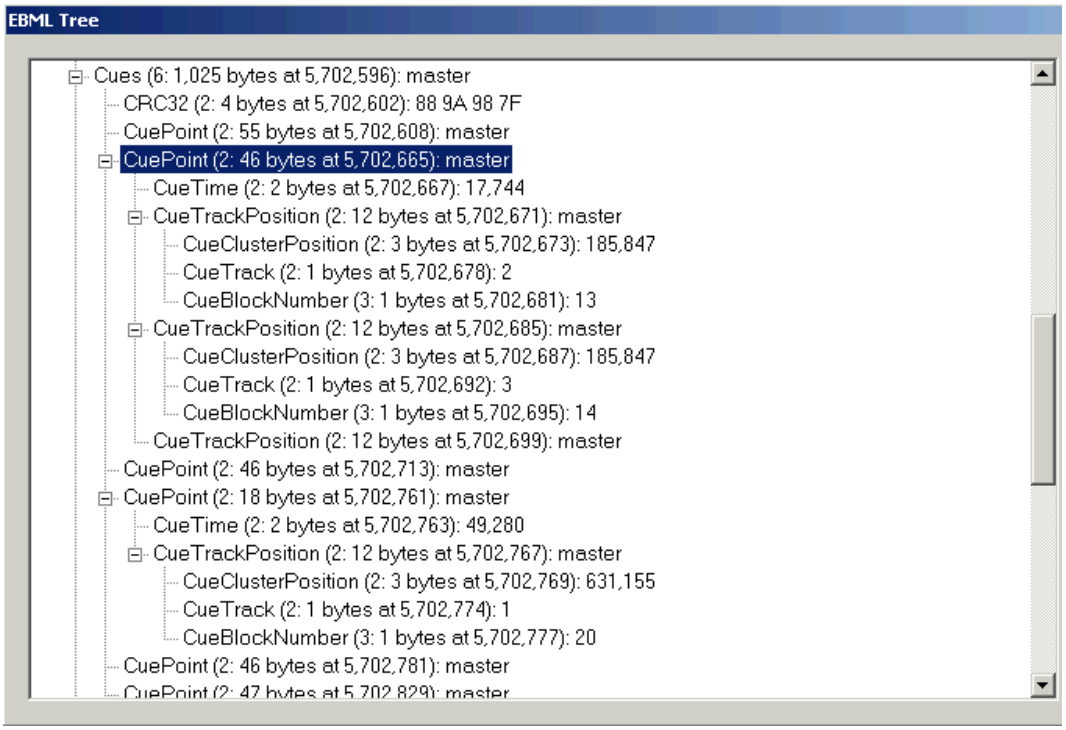
  - the difference
    `<timecode of next block of the same stream> - <timecode>`
  - equal to `DefaultDuration` (for the last block of each stream)

  As a consequence, the `Duration` element is mandatory for every `block` of subtitle tracks.

- **Block**
  contains data to be replayed. See page 26 for details.

5.5 Cues

The `Cues` element contains information helpful (but not necessary) for seeking. Each piece of information, called a `CuePoint`, contains a timestamp, and a list of pairs (track number, (cluster position[, block number within cluster])). Generally, a `CuePoint` should only point to keyframes.



5.5.1 CueTime (int)

The `CueTime` element contains a scaled timecode, telling which timestamp this `CuePoint` is pointing to.

5.5.2 CueTrackPositions

One `CueTrackPositions` element contains information where to find the block with timecode `CueTime` of a certain track:

- **CueClusterPosition (int)**

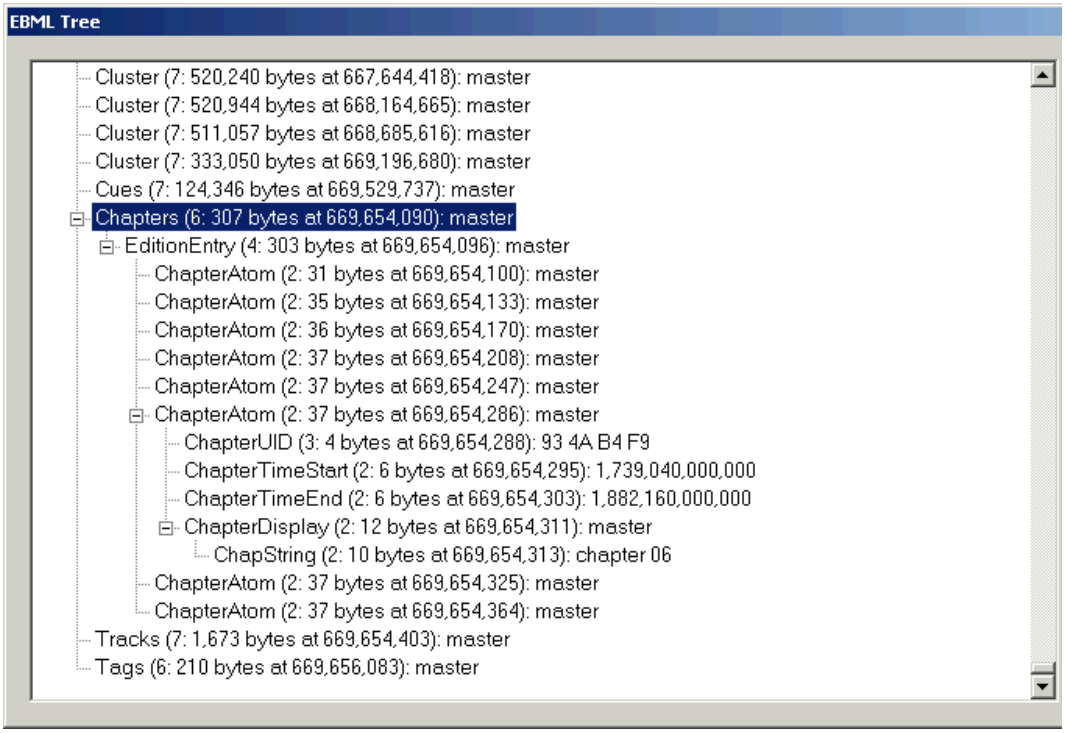
Defines the position of the cluster containing the refered block within the segment.

- **CueTrack (int)**
  Defines which Track the refered block belongs to

- **CueBlockNumber (int)**
  Number of Block in the specified cluster. The first block is number 1. `CueBlockNumber` is not mandatory.

5.6 Chapters

The `Chapters` element contains a list of all chapters found in this `Segment`. The basic structure is indicated here:



Each Chapter is decribed by one `ChapterAtom`, and all `ChapterAtoms` are children of `EditionEntry` elements.
Each `ChapterAtom` contains

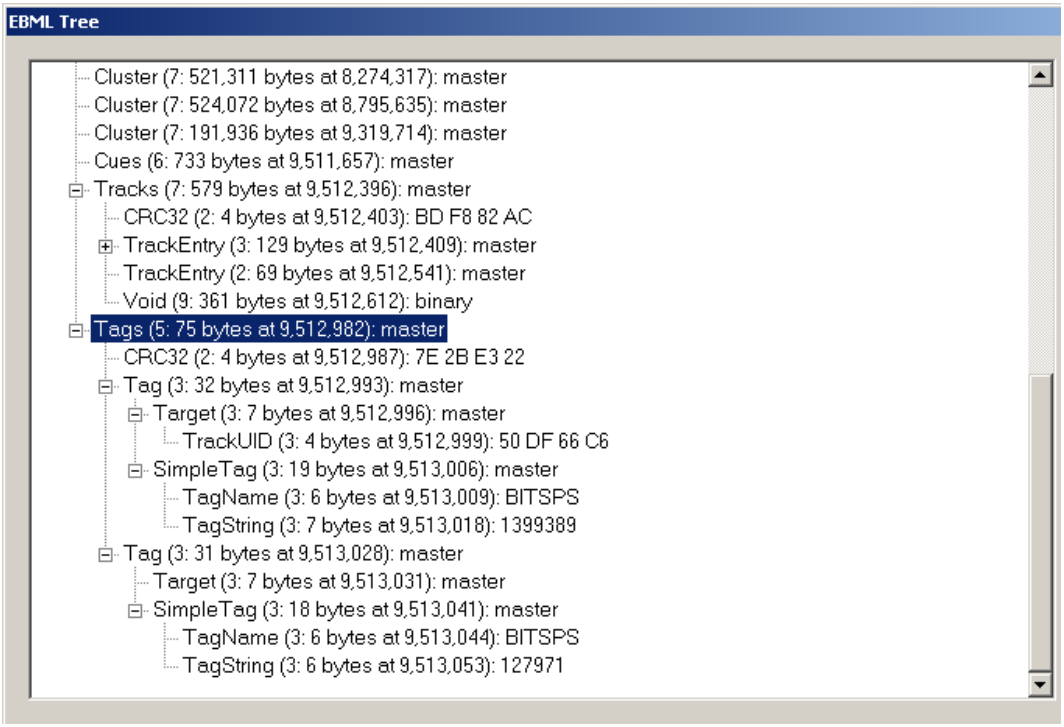
- **ChapterUID**
  A 32 bit number identifying this chapter uniquely. ChapterUID and TrackUID numbers shall not collide.
- **ChapterFlagHidden** If this value is 1, the chapter is hidden. In this case, the user interface should not allow to select this chapter. It should only be available to control tracks.
- **ChapterFlagEnabled** If this value is 0, the chapter is disabled. The player should skip the entire chapter in this case.

- **ChapterTimeStart** Unscaled (!) timecode of the beginning of the chapter.
- **ChapterTimeEnd** Unscaled (!) timecode of the end of the chapter. If `ChapterTimeEnd` is not indicated, the corresponding chapter ends
  - at the beginning of the next chapter
  - at the end of the parent chapter, if the chapter is a subchapter
  - at the end of the segment, if the chapter is not a subchapter
- **ChapterDisplay** contains names for the chapter. You can write several `ChapterDisplay` elements if you want to store different names in different languages for one chapter.
  - **ChapterString**: Name of chapter
  - **ChapterLanguage**: Language of the value of `ChapterString`
  - **ChapterCountry**: The country corresponding to `ChapterString`
- **ChapterAtoms**: Chapters which are subchapters to the current chapter.

5.7 Tags

`Tags` provide additional information not important for replay. A complete list of specified tags can be found at
www.matroska.org

A `Tags` element contains a number of `Tag` elements. Each `Tag` element contains a list of UIDs (usually `TrackUID`s or `ChapterUID`s), and a list of `SimpleTag`s, each one containing a name and a value:



If no `Target`s are specified, then the `Tag` is a global `Tag` refering to the entire `Segment`.

6 *Matroska* block Layout and Lacing

6.1 Basic layout of a Block

A *Matroska* block has the following format:

```
BLOCK {
  vlint     TrackNumber
  sint16    Timecode  // relative to Cluster timecode
  int8      Flags     // gap, lacing
  if (lacing) {
    int8      frame_count-1
    if (lacing ==  EBML lacing) {
       vlint     size[0]
       svlint    size[1..frame_count-2]
    } else
    if (lacing == Xiph lacing) {
      int8       size[size of <leading (frame_count-1) frames> / 255 + 1]
    }
  }
  int8[]    data
}
```

The following bits are defined for `Flags`:

```
  Bit 7  : gap (last block of track)
  Bit 5-6: lace type
              00 - no lacing
              01 - Xiph lacing
              11 - EBML lacing
              10 - fixed-size lacing
```

The type of lacing in use defines how the `size` values are to be read.

6.2 Lacing

Lacing is a technique that allows to store more than one atom of data (like one audio frame) in one block, with the goal to decrease overhead, without losing the ability to separate the frames in a lace later again.

Generally, the size of the last frame in a Lace is not stored, as it can be derived from the total block size, the size of the block header and the sum of the sizes of all other frames.
Frame duration values are not preserved! That means, it is highly recommended **not** to use lacing if the frame duration is not constant, like Vorbis audio.

6.2.1 Xiph Lacing

The size of each frame is coded as a sum of int8. A value smaller than 255 indicates that the next value refers to the next frame.

Example
`size = { 187, 255, 255, 120, 255, 0, 60 }` means that there are 4 frames with `187, 630, 255, 60` bytes.

6.2.2 EBML Lacing

Size of first frame ("frame `0`") of a lace = `size[0]`
Size of frame `i` of a lace: `size[i] - size[i-1]`

6.2.3 Fixed Lacing

Fixed Lacing is used if all frames in a lace have the same size. Examples are AC3 or DTS audio. In this case, knowing the number of frames is enough to calculate the size of one frame. Consequently, there are no `size` values.

7 Overhead of *Matroska* files

The scope of this section is explaining how to predict the overhead of a *Matroska* file before muxing, and without analysing any of the source files excessively.

7.1 Overhead of `BlockGroups`

First, here again the layout of a typical `BlockGroup`

```
BlockGroup <size>
   Block <size> <number, flag, timecode>
   [ Reference <size> <val> ]
```

The EBML identication for `Block`s and `BlockGroup`s are 1 byte each, so that the structure above, not counting `Reference`s, takes:

- BlockGroup $<$ 128 bytes: **8 bytes**
- BlockGroup $<$ 16kbytes: **10 bytes**
- BlockGroup $<$ 2MBytes: **12 bytes**

`BlockGroup`s larger than 2MBytes are extremely unlike, and even `BlockGroup`s larger than 16kBytes won't occur often, compared to `BlockGroup`s between 128 bytes and 16 kBytes. That means, assuming an overhead of 10 bytes for `BlockGroup`s without `References` usually results in a good approximation.

7.1.1 video

In a typical video stream, there are a lot of frames with 1 `Reference` (P-Frames, Delta-Frames), and a few keyframes. Typical rations are 100:1. There might also be frames with 2 `References` (B-Frames), e.g. native MPEG4 streams. Assuming a ratio of 66:33:1 for B:P:K, and assuming a bitrate far below 3,2 MBit/s (meaning that typical B- and P-frames are smaller than 16 kB), that causes about 15 bytes of overhead per frame. If there are no B-Frames, there are about 13 bytes per frame.

Example: 2 hours, 25 fps.
The video stream will cause around 2,3 MB of overhead.

7.1.2 audio - without lacing

As audio does usually not have any `References` (all audio frames are keyframes), one audio frame will take 8 or 10 bytes of overhead. For MP3, AC3, DTS and AAC, frames causing 8 bytes of overhead are unlikely. They are more likely for Vorbis.

Example: MP3 audio, 24ms per frame, duration: 2h
This stream will cause 3MB of overhead.

7.1.3 audio - with lacing

1. CBR+CFR: fixed lacing
In this case, *fixed lacing* (see section 6.2.3) is used. With fixed lacing, the overhead is the normal `BlockGroup` overhead, plus 1 byte for the lace header. Assuming that `BlockGroups` are not larger than 16k, that means that the overhead per frame is equal to `11 / frame_count`

Example: AC3 audio, 448 kbps, 1792 bytes per frame, 32ms per frame
1.) 8 frames per lace.
overhead for one frame = `11/8 = 1,375` bytes = `1 byte / 23,3 ms`.
2.) 9 frames per lace.
overhead for one frame = `11/9 = 1,222` bytes = `1 byte / 26,2 ms`.
3.) 10 frames per lace.
overhead for one frame = `13/10 = 1,3` bytes = `1 byte / 24,6 ms`.

An AC3 stream of 2 hours with 9 frames per lace will cause 270kB of overhead.

2. no CBR, but almost all frames smaller than 255 bytes: XIPH lacing
In this case, XIPH lacing (see section 6.2.1) is used, meaning that the overhead of a `BlockGroup` is equal to `normal BlockGroup overhead + frame_count`, meaning that the overhead per frame is about `(11+frame_count)/frame_count`, if there are `frame_count` frames in each lace. Again, if the `BlockGroups` are larger than 16kBytes, then the overhead is `(13+frame_count)/frame_count`.
In other words, the ratio in `bytes / frame` will always be between about `1,2` and `2,5` for audio streams with mainly small frames.

Although XIPH lacing is also defined for larger frames, EBML lacing is usu-

ally more effective then.

3. otherwise: EBML lacing Assuming that the difference in size between 2 consecutive frames is smaller than 8191, 1 or 2 bytes are needed to code the size of each frame, additionally to the normal `BlockGroup` overhead.

As a result, we get 3 possible estimations:

a) worst case That means, a lace with `frame_count` frames using EBML lacing will cause not more than `((11 or 13)+2*frame_count)/frame_count` bytes of overhead per frame.

Example 1: 16 frames per lace, `BlockGroup` > 16kB, worst case:
`overhead <= (13 + 2*16)/16 = 2,8 bytes / frame`.
Example 2: 8 frames per lace, `BlockGroup` < 16kB, worst case:
`overhead <= (11 + 2*8)/8 = 3,4 bytes / frame`.

b) best case The best case is obviously that 2 consecutive frames differ by not more than 62 bytes. In that case, one byte is needed to code the size of one frame. However, the first frame might need to bytes, if it is larger than 126 bytes.

Example 1: 16 frames per lace, `BlockGroup` > 16kB, best case:
`overhead <= (13 + 1*16)/16 = 1,8 bytes / frame`.
Example 2: 8 frames per lace, `BlockGroup` < 16kB, best case:
`overhead <= (11 + 1*8)/8 = 2,4 bytes / frame`.

c) average case This is the case you need for optimal overhead prediction. Unfortunately, the average case depends on the compression format of the corresponding audio track, its bitrate, maybe even the encoder that has been used. The easiest way to gather data on the average case of EBML lace header overhead is to simulate the lace results of different files that are likely to be used. Candidates are MPEG 1/2/4 audio and Vorbis, but not AC3 or DTS.
I have run a simulation with the following file types:
MPEG 1 Layer 3 (128 and 192 kbps, 48 kHz), HE-AAC (224 kbps and 96 kbps, 44,1 kHz), LC-AAC (268 kbps, 44,1 kHz)

The results obtained from those files are discussed on the following pages. The lace behaviour simulation has been run using `mls` (short for 'matroska lace simulator'), which can be found on

www-user.tu-chemnitz.de/~noe/Video-Zeug

Note that it would be required to run the simulation and to evaluate the results as follows for each audio format, in each bitrate, maybe even with each encoder, for which results as accurate as possible shall be predicted.

The results for the lace header size are as follows:

	Lace header overhead per frame @ <x> Frames per lace								
	4	8	12	16	24	32	48	64	96
Audio format									
MP3 @128	1,39	1,29	1,26	1,24	1,22	1,22	1,21	1,20	1,20
MP3 @192	1,50	1,41	1,38	1,37	1,36	1,35	1,34	1,34	1,33
HE-AAC@224	1,39	1,29	1,25	1,24	1,22	1,21	1,20	1,20	1,20
HE-AAC@ 64	1,34	1,23	1,19	1,18	1,16	1,15	1,14	1,14	1,13
LC-AAC@268	1,31	1,19	1,16	1,14	1,12	1,11	1,10	1,09	1,09

Applications using `libmatroska` for *Matroska* file creation are using 8 frames per lace. As a consequence, the overhead for a track using EBML lacing can be predicted to an acceptable accuracy if the audio format is known.
As you can also see, larger laces hardly affect the overhead caused by the lace headers of `Block`s from a certain size on.

However, larger laces mean less `Block`s and thus less `BlockGroup`s, so the total overhead per frame, including the overhead caused by overhead outside of the `Blocks`, is worth a look. Here are the results with the same test files as above

	Overhead per frame @ <x> Frames per lace								
	4	8	12	16	24	32	48	64	96
Audio format									
MP3 @128	4,14	2,67	2,17	1,93	1,68	1,56	1,48	1,41	1,33
MP3 @192	4,25	2,79	2,30	2,06	1,81	1,75	1,61	1,54	1,47
HE-AAC@224	4,14	2,66	2,23	2,05	1,76	1,62	1,48	1,40	1,33
HE-AAC@ 64	4,09	2,61	2,11	1,86	1,62	1,49	1,40	1,34	1,27
LC-AAC@268	4,06	2,57	2,07	1,82	1,66	1,51	1,37	1,30	1,22

Now lets take the 2nd table and find out how much overhead that means in a real movie of 2 hours.
In the case of the mp3 files used in that example, one frame lasts 24ms. In the case of our LC-AAC file, one frame lasts 23,22 ms, and for the HE-AAC

file we get 46,44ms.
Thus a file of 2 hours will have the following number of frames:
MP3 - 300,000
LC-AAC - 310,000
HE-AAC - 155,000.

First, lets use the default setting of `libmatroska` (8 frames per lace) and calculate the overhead a muxing app using `libmatroska` would cause when muxing those files into a movie:

- **MP3 @ 128**: overhead = 300,000 * 2,67 = 801,000 bytes
- **MP3 @ 192**: overhead = 300,000 * 2,79 = 837,000 bytes
- **HE-AAC @ 224**: overhead = 155,000 * 2,66 = 412,300 bytes
- **LC-AAC @ 268**: overhead = 310,000 * 2,57 = 796,700 bytes

With 24 frames per lace, an MP3 block would have a duration of 576ms, an HE-AAC block even about 1 second. That means, when seeking in a file, an awkward impression of the audio being missing for a moment could occur. Thus, larger laces than 1 second are highly discouraged. Nethertheless, let's analyze the overhead in our file for laces of 24 and 96 frames each, and compare the overhead to the one caused by `libmatroska`. Here is the corresponding table:

	Frames per lace		
Audio format	8	24	96
MP3 @128	782kB	492kB	389kB
MP3 @192	817kB	530kB	430kB
HE-AAC@224	402kB	266kB	201kB
HE-AAC@ 64	395kB	245kB	192kB
LC-AAC@268	778kB	502kB	369kB

As you can see, putting 24 frames in one block, compared to 8 frames, saves some overhead. However, putting 96 frames in one `Block` instead of 24 saves less overhead than 24 compared to 8. As 96 frames per lace will usually cause uncomfortable seeking, it is recommended not to put more than about 24 frames in one `Block`.

7.2 Overhead of `Clusters`

Although most of the overhead is caused by `BlockGroup`s, the amount of overhead caused by `Clusters` themselves is noticeable as well.

Here again the basic layout of a `Cluster`:

```
Cluster <size>
  [ CRC32 ]
  TimeCode <size> <timecode>
  [ PrevClusterSize <size> <prevsize> ]
  [ Position <size> <position> ]
  { BlockGroup }
```

First, some conventions:

- each `Cluster` has a size between 16kB and 2MB
- each `Cluster` may begin between 16MB and 4GB

As typical movie files are designed to fit on 1 or 2 CDs, or 2 or 3 of them fill one DVD, point 2 will be true for most of the clusters in typical files.

With the abovementioned restrictions on `Cluster`s, the overhead inside one Cluster will be:

- `Cluster` ID + <size>: 7 bytes
- `CRC32`: 6 bytes
- `Timecode`: 5 bytes
- `PrevClusterSize`: 5 bytes
- `Position`: 5 bytes
- `Seekhead` entry for `Cluster`: 17 bytes

Depending on the muxing settings, the overhead caused by one `Cluster` will be between 12 and 45 bytes.

Example: Assuming a size of 1 MB per `Cluster`, that means an overhead rate of 0,001% - 0,005%, or up to 100 kB in a file of 2GB.

7.3 Overhead caused by `Cues`

Here again the layout of a `CuePoint`:

```
CuePoint <size>
  CueTime <size> <time>
  { CueTrackPosition <size>
      CueClusterPosition <size> <position>
      CueTrack <size> <track>
      [ CueBlockNumber <size> <block number> ]
  }
```

Assuming that a `CuePoint` only points into one certain track, the overhead is:

- CuePoint: 2 bytes
- CueTime: 5 bytes
- CueTrackPosition: 2 bytes
- CueClusterPosition: 6 bytes
- CueTrack: 3 bytes
- CueBlockNumber: 4 bytes

Total: `22 bytes`.

Example: Assuming that there is a `CuePoint` each 4 seconds (1 keyframe in 100 frames), this adds on overhead of `0,22 bytes / frame`

There can also be `CuePoint`s for audio tracks. In that case, as every frame will be a keyframe, the number of `CuePoint`s only depends on the muxing application. Predicting the overhead requires to know its behaviour.

8 Questions, Comments, Contact, Links

8.1 Links

Matroska pages:

`http://www.matroska.org`

8.2 Questions, Comments, Contact

If you have any questions concerning this document, if you have comments, additions, if you have found an error, or if you want to contact me for whatever reason, send a mail (include 'matroska' in the topic!) to

`noe@hrz.tu-chemnitz.de`

in german, english or french.