Matroska File Format (under construction!)

Alexander Noé
alex@alexander-noe.com

Last change: November 28, 2009

Contents

1 **Introduction** 4

2 **EBML - basics** 6

2.1 Unsigned Integer Values of Variable Length (”vint“) 6
2.2 EBML elements 7
2.3 Signed Integer Values of Variable Length (svint) 7
2.4 Data Types 8

3 **MATROSKA files - Top-Level elements** 9

3.1 EBML 9
3.2 Segment 9

4 **EBML - The EBML file header** 10

5 **Level 1 - Elements inside Segments** 12

5.1 Overview 12
5.2 SegmentInfo 13
5.3 SeekHead 16
5.4 Tracks 18
5.5 Cluster 27
5.6 Cues 29
5.7 Chapters - Editions and ChapterAtoms 31
5.8 Attachments 35
5.9 Tags 36

6 MATROSKA block Layout and Lacing **39**

6.1 Basic layout of a Block 39

6.2 Lacing 40

7 Overhead of MATROSKA files **41**

7.1 Overhead of **BLOCKGROUPS** 41

7.2 Overhead of **CLUSTER**S 46

7.3 Overhead caused by `Cues` 47

8 Links **48**

List of Tables

1 EBML 10
2 Segment 12
3 SegmentInfo 14
4 SeekHead 16
5 Seek 16
6 Tracks 18
7 TrackEntry 18
8 Video 21
9 Audio 22
10 ContentEncodings 23
11 ContentEncoding 23
12 ContentCompression 24
16 Cluster 27
17 BlockGroup 28
18 Cues 29
19 CuePoint 30
20 CueTrackPositions 30
21 Chapters 31
22 EditionEntry 31
23 ChapterAtom 32

24 ChapterTracks 34
25 ChapterDisplay 34
26 Attachments 35
27 AttachedFile 35
28 Tags 36
29 Tag 37
30 Targets 37
31 SimpleTag 38

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, the usage of existing libraries isn't.

This document does not replace the official documentation[1]. It is less condensed, but not necessarily complete. Especially, in the case that MATROSKA supports Digital Restrictions Management one day, I will expressively not document that part. Also, typos in element IDs are never impossible.

When speaking about element occurence, elements can be mandatory or not, elements may be present several times inside a parent element or not etc. Occurence restrictions will be indicated using expressions like $= 1$ or ≥ 1 etc. Those restrictions will exclude cases which do not technically render a file unusable or ambigous, but which are unreasonable, like a file with no **SEGMENTUID**, see section 5.2. The same way it would be weird (but not make a file unusable) to have a **CHAPTERS** element (which is supposed to describes chapters) which is empty. An element that must occur at least once is a reasonable file is called "mandatory". When an element is really mandatory, i.e the file or a part of it is useless when it's missing, it will be labeled as ≥ 1 (!) or $= 1$ (!). An example would be the codec ID of a track, without which a track cannot be decoded at all.

The official Matroska specification pages use the following interpretation of "mandatory" and "default": When an element has a default value that is used if the element itself is not present, the value cannot be missing, thus the element is inherently mandatory. This interpretation of "mandatory" being weird, this document considers an element mandatory when it must be physically present in the file. Also, default values can only be valid values. Consequently, a mandatory element cannot have a default value because if it had one, it couldn't be mandatory anymore.

In this document, element names are always printed like **THIS**, element values are printed like **$THIS**, as in "if **$THISFLAG**=1, ...".

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,

---

[1] `http://www.matroska.org/technical/specs/index.html`

please send me an e-mail (include ’matroska’ in the topic!). You can contact me in german, english or french, whatever you prefer. Just don’t ask me if you can ask something or if I could document some Digital Restrictions Management.

This document is powered by LaTeX, so changing the order of certain tables or the style of those tables etc. is, with certain limits, possible within a few seconds.

Screenshots of real life file structures are used to illustrate the file structure. All of them have been made using the EBML Tree Viewer in AVI-Mux GUI.

2 EBML - basics

EBML files use integers of variable size. This way, the file format doesn't waste space with storing 32 or even 64 bit integers in placed where they *might sometimes* occur. The way the size is coded is inspired by the UTF-8 encoding format.

2.1 Unsigned Integer Values of Variable Length (”vint“)

The length of an integer is equivalent to $length = 1 + [number\_of\_leading\_zero\_bits]$. All integers use big endian. You could use more than 7 leading zeros, then the first byte would be 0x00, however, this would only be needed if integers longer than 56 bits are required. This is forbidden in MATROSKA files.

Example: 3A 41 FE:

The first byte 3A (0011 1010) has 2 leading zeros, resulting in a total length of 3 bytes. The first '1' in the byte (0011 1010) is just needed to finish the sequence of leading zeros and can't be used to store the value either. Thus, it is reset to obtain the value this byte sequence represents. The result is then 0x1A41FE. As you can see, you lose one bit per byte to know how long a number is, and you can use 7 bits per byte to store the integer's value itself.

Of course, the value 0x1A41FE could also be written as 10 1A 41 FE or 08 00 1A 41 FE (do the decoding on a piece of paper if it's not clear), however, when writing EBML files, the shortest possible encoding should be used to avoid wasting space, which is the very point of this coding scheme.

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 must be encoded again with a greater destination length. Example: When encoding 16383 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 EBML elements

One piece of information is stored the following way:

```
typedef struct {
    vint      ID        // EBML-ID
    vint      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.

2.3 Signed Integer Values of Variable Length (svint)

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,
      0x00FFFFFFFFFFFF, 0x007FFFFFFFFFFFFF };
```

2.4 Data Types

Whereas `vints` are used in the header section of EBML elements, the data types describes in this section occur in the data section.

2.4.1 Signed and Unsigned Integers (int and uint)

Integers, signed as well as unsigned, are stored in big endian byte order, with leading 0x00 (in case of positive values) and 0xFF (in case of negative values) being cut off (example for int: -257 is 0xFE 0xFF). An int/uint may not be larger than 8 bytes.

2.4.2 Float

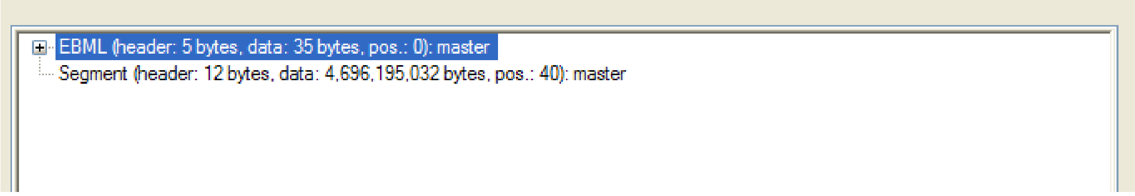
A `Float` value is a 32 or 64 bit real number, as defined in IEEE. 80 Bit values have been in the specification, but have been removed and should not be used. The bytes are stored in big endian order.

2.4.3 Types of Strings

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

3 MATROSKA files - Top-Level elements

MATROSKA files only have two different top level elements:



EBML (header: 5 bytes, data: 35 bytes, pos.: 0): master
Segment (header: 12 bytes, data: 4,696,195,032 bytes, pos.: 40): master

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.

3.2 Segment

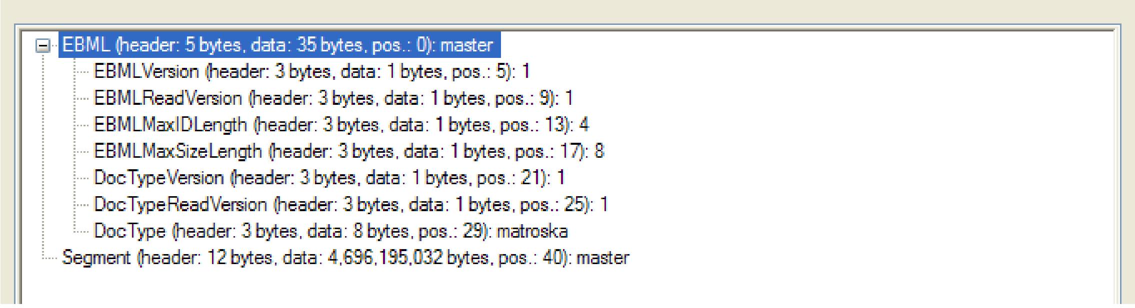
A **SEGMENT** contains multimedia data, as well as any header data necessary for replay. There can be several **SEGMENTS** in one MATROSKA file, but this is not encouraged to be done, as not many tools are able to handle multisegment MATROSKA files correctly. If you want to replay multisegment MATROSKA files on Windows, please use Haali Media splitter[2]

---

[2] `http://haali.cs.msu.ru/mkv/`

4 EBML - The EBML file header

The **EBML** top level element contains a description of the file type, such as EBML version, file type name, file type version etc.



```
EBML (header: 5 bytes, data: 35 bytes, pos.: 0): master
    EBMLVersion (header: 3 bytes, data: 1 bytes, pos.: 5): 1
    EBMLReadVersion (header: 3 bytes, data: 1 bytes, pos.: 9): 1
    EBMLMaxIDLength (header: 3 bytes, data: 1 bytes, pos.: 13): 4
    EBMLMaxSizeLength (header: 3 bytes, data: 1 bytes, pos.: 17): 8
    DocTypeVersion (header: 3 bytes, data: 1 bytes, pos.: 21): 1
    DocTypeReadVersion (header: 3 bytes, data: 1 bytes, pos.: 25): 1
    DocType (header: 3 bytes, data: 8 bytes, pos.: 29): matroska
Segment (header: 12 bytes, data: 4,696,195,032 bytes, pos.: 40): master
```

Obviously, this header being missing makes it necessary to guess the file type.

Table 1: The EBML element (Top-Level)

Element	Description
uint, # ≤ 1 **EBMLVERSION** ID: `42 86` def: 1	indicates the version of the EBML Writer that has been used to create a file
uint, # ≤ 1 **EBMLREADVERSION** ID: `42 F7` def: 1	indicates the minimum version an EBML parser needs to be compliant with to be able to read the file
uint, # ≤ 1 **EBMLMAXIDLENGTH** ID: `42 F2` def: 4	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.
uint, # ≤ 1 **EBMLMAXSIZELENGTH** ID: `42 F3` def: 8	indicates the maximum `s_size` value the file contains. Any EBML element having an `s_size` value greater than EBMLMaxSizeLength shouldl be considered invalid.

EBML *continued on next page*

Element	Description
string, # ≤ 1 **DOCTYPE** ID: `42 82` def: matroska	describes the contents of the file. In the case of a MATROSKA file, its value is `'matroska'`
uint, # ≤ 1 **DOCTYPEVERSION** ID: `42 87` def: 1	indicates the version of the **$DOCTYPE** writer used to create the file
uint, # ≤ 1 **DOCTYPEREADVERSION** ID: `42 85` def: 1	indicates the minimum version number a **$DOCTYPE** parser must be compliant with to read the file.

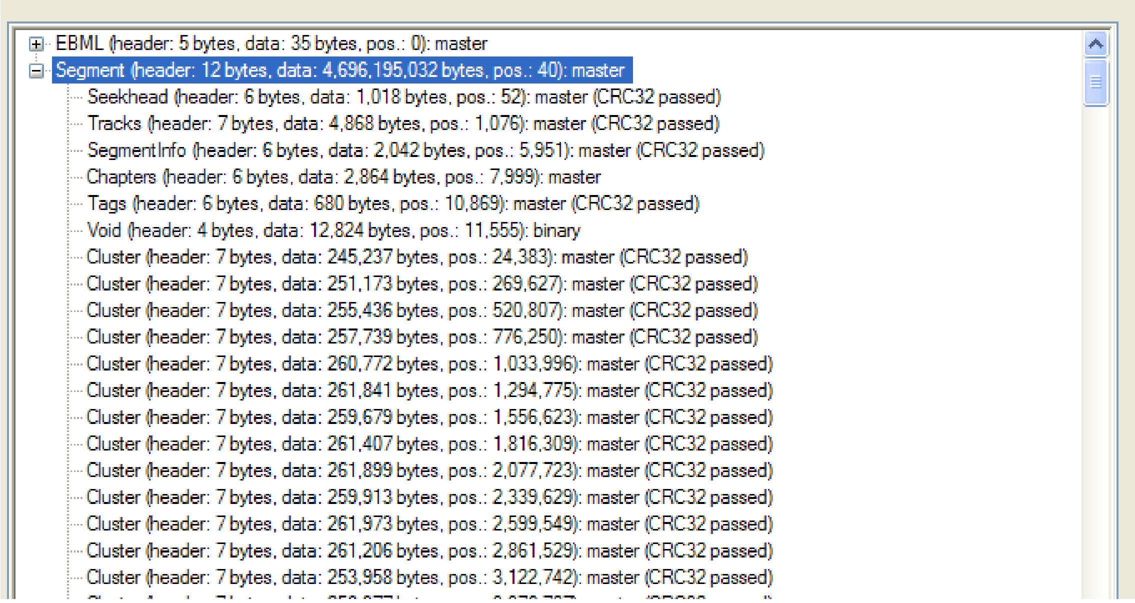
Index →page 2 *end of* **EBML**

As you can see, in the case of Matroska files all child elements of the **EBML** element have a default value. Thus, an empty **EBML** element would technically introduce a Matroska file (with file type version 1, maximum ID length 4, maximum size length 8 etc.) correctly. However, I don't recommend to push the specifications like this.

It is not recommended to use either IDs or size values greater than 8 bytes. While it's clear that 8 bytes are enough to represent any size of anything on any hard disc, one might think about using IDs larger than 8 bytes. However, since the ID is considered an integer, treating IDs larger than 8 bytes is difficult on current CPUs, which are limited to 64 bit for simple integer operations.

5 Level 1 - Elements inside Segments

5.1 Overview



```
EBML (header: 5 bytes, data: 35 bytes, pos.: 0): master
Segment (header: 12 bytes, data: 4,696,195,032 bytes, pos.: 40): master
    Seekhead (header: 6 bytes, data: 1,018 bytes, pos.: 52): master (CRC32 passed)
    Tracks (header: 7 bytes, data: 4,868 bytes, pos.: 1,076): master (CRC32 passed)
    SegmentInfo (header: 6 bytes, data: 2,042 bytes, pos.: 5,951): master (CRC32 passed)
    Chapters (header: 6 bytes, data: 2,864 bytes, pos.: 7,999): master
    Tags (header: 6 bytes, data: 680 bytes, pos.: 10,869): master (CRC32 passed)
    Void (header: 4 bytes, data: 12,824 bytes, pos.: 11,555): binary
    Cluster (header: 7 bytes, data: 245,237 bytes, pos.: 24,383): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 251,173 bytes, pos.: 269,627): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 255,436 bytes, pos.: 520,807): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 257,739 bytes, pos.: 776,250): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 260,772 bytes, pos.: 1,033,996): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 261,841 bytes, pos.: 1,294,775): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 259,679 bytes, pos.: 1,556,623): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 261,407 bytes, pos.: 1,816,309): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 261,899 bytes, pos.: 2,077,723): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 259,913 bytes, pos.: 2,339,629): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 261,973 bytes, pos.: 2,599,549): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 261,206 bytes, pos.: 2,861,529): master (CRC32 passed)
    Cluster (header: 7 bytes, data: 253,958 bytes, pos.: 3,122,742): master (CRC32 passed)
```

Table 2: The SEGMENT element (Top-Level)

Element	Description
Master, # $= 1$ **SEGMENTINFO** (→3) ID: `15 49 A9 66`	**SEGMENTINFO** contains general information about a segment, like an UID, a title etc. This information is not really required for playback, but should be there (→ section 5.2).
Master, # ≥ 0 **SEEKHEAD** (→4) ID: `11 4D 9B 74`	A **SEEKHEAD** is an index of elements that are children of **SEGMENT**. It can point to other **SEEKHEAD**s, but not to itself. If all non-**CLUSTER** precede all **CLUSTER**s (→ section 5.5), a **SEEKHEAD** is not really necessary, otherwise, a missing **SEEKHEAD** leads to long file loading times or the inability to access certain data.
Master, # ≥ 0 **CLUSTER** (→16) ID: `1F 43 B6 75`	A **CLUSTER** contains video, audio and subtitle data. Note that a MATROSKA file could contain chapter data or attachments, but no multimedia data, so **CLUSTER** is not a mandatory element.

SEGMENT *continued on next page*

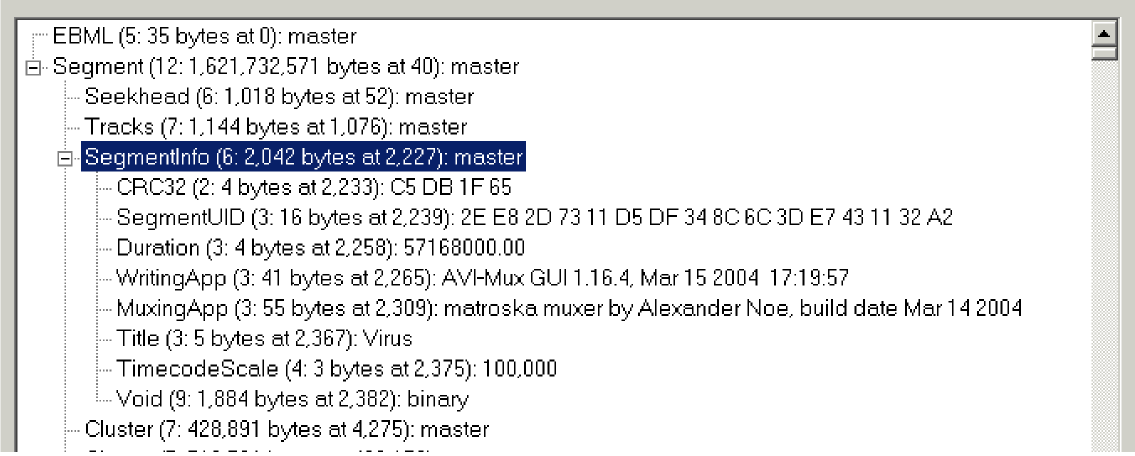
Element	Description
Master, # ≥ 0 **TRACKS** (→6) ID: `16 54 AE 6B`	A **TRACKS** element contains the description of some or all tracks (preferably all). This element can be repeated once in a while for backup purposes. A file containing only chapters and attachments does not have a **TRACKS** element, thus it's not mandatory.
Master, # ≤ 1 **CUES** (→18) ID: `1C 53 BB 6B`	The **CUES** element contains a timestamp-wise index to **CLUSTERS**, thus it's helpful for easy and quick seeking.
Master, # ≤ 1 **ATTACHMENTS** (→26) ID: `19 41 A4 69`	The **ATTACHMENTS** element contains all files attached to this **SEGMENT**.
Master, # $= 1$ **CHAPTERS** (→21) ID: `10 43 A7 70`	The **CHAPTERS** elements contains the definition of all chapters and editions of this **SEGMENT**
Master, # ≤ 1 **TAGS** (→28) ID: `12 54 C3 67`	The **TAGS** element contains further information about the **SEGMENT** or elements inside the **SEGMENT** that is not really required for playback.

Index →page 2

end of **SEGMENT**

5.2 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.



```
EBML (5: 35 bytes at 0): master
Segment (12: 1,621,732,571 bytes at 40): master
    Seekhead (6: 1,018 bytes at 52): master
    Tracks (7: 1,144 bytes at 1,076): master
    SegmentInfo (6: 2,042 bytes at 2,227): master
        CRC32 (2: 4 bytes at 2,233): C5 DB 1F 65
        SegmentUID (3: 16 bytes at 2,239): 2E E8 2D 73 11 D5 DF 34 8C 6C 3D E7 43 11 32 A2
        Duration (3: 4 bytes at 2,258): 57168000.00
        WritingApp (3: 41 bytes at 2,265): AVI-Mux GUI 1.16.4, Mar 15 2004  17:19:57
        MuxingApp (3: 55 bytes at 2,309): matroska muxer by Alexander Noe, build date Mar 14 2004
        Title (3: 5 bytes at 2,367): Virus
        TimecodeScale (4: 3 bytes at 2,375): 100,000
        Void (9: 1,884 bytes at 2,382): binary
    Cluster (7: 428,891 bytes at 4,275): master
```

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

Table 3: The SegmentInfo element, child of Segment (→2)

Element	Description
char[16], # = 1 **SegmentUID** ID: `73 A4`	a unique 128 bit number identifying a **Segment**. Obviously, a file can only be referred to by another file if a **SegmentUID** is present, however, playback is possible without that UID.
utf-8, # ≤ 1 **SegmentFilename** ID: `73 84`	contains the name of the file the **Segment** is stored in. Since renaming files is easy, the reliability of this element's value should not be overrated.
char[16], # ≤ 1 **PrevUID** ID: `3C B9 23`	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 the user tries to seek to a timecode earlier than the earliest timecode of the active **Segment**. That **segment** should, of course, be easy to locate, for instance in a file in the same directory.
utf-8, # ≤ 1 **PrevFilename** ID: `3C 83 AB`	contains the name of the file in which the **Segment** having the ID **$PrevUID** is stored. **PrevFilename** should not be considered reliable for the same reason as **SegmentFilename**, however, it could be the first filename the player is looking for when the **segment** described in **PrevUID** is needed

SegmentInfo *continued on next page*

Element	Description
char[16], # ≤ 1 **NEXTUID** ID: `3E B9 23`	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 the user tries to seek to a timecode after the end of the active **SEGMENT**. Like **PREVUID**, the corresponding **SEGMENT** should be easy to locate.
utf-8, # ≤ 1 **NEXTFILENAME** ID: `3E 83 BB`	contains the name of the file in which the **SEGMENT** having the ID **$NEXTUID** is stored. **NEXTFILENAME** shall not be considered reliable for the same reason as **SEGMENTFILENAME**.
uint, # ≤ 1 **TIMECODESCALE** ID: `2A D7 B1`	Each scaled timecode in a MATROSKA file is multiplied by **TIMECODESCALE** to obtain a timecode in nanoseconds. Note that not all timecodes are scaled!
float, # ≤ 1 **DURATION** ID: `44 89`	The **DURATION** indicates the duration of the **SEGMENT**. The duration measured in nanoseconds is scaled and is thus equal to **$DURATION** * **$TIMECODESCALE**. This element should be written.
utf-8, # ≤ 1 **TITLE** ID: `7B A9`	Contains a general name of the **SEGMENT**, like ''`Lord of the Rings - The Two Towers`''. No language can be attached to the title, however, Tags (→ section 5.9) could be used to define several titles for a segment. This is not yet commonly done, though.
string, # $= 1$ **MUXINGAPP** ID: `4D 80`	contains the name of the library that has been used to create the file (like "libmatroska 0.7.0"). This element should be written by any muxer! Especially if non-compliant files are encountered, this help to know who must be blamed for that file.
utf-8, # $= 1$ **WRITINGAPP** ID: `57 41`	contains 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**.
int, # ≤ 1 **DATEUTC** ID: `44 61`	contains the production date, measured in nanoseconds relatively to Jan 01, 2001, 0:00:00 GMT+0h

Index →page 2 *end of* **SEGMENTINFO**

5.3 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:

Table 4: The SEEKHEAD element, child of SEGMENT (→2)

Element	Description
Master, # ≥ 1 **SEEK** (→5) ID: `4D BB`	One **SEEK** element contains an EBML-ID and the position within the **SEGMENT** at which an element with this ID can be found.

Index →page 2 *end of* **SEEKHEAD**

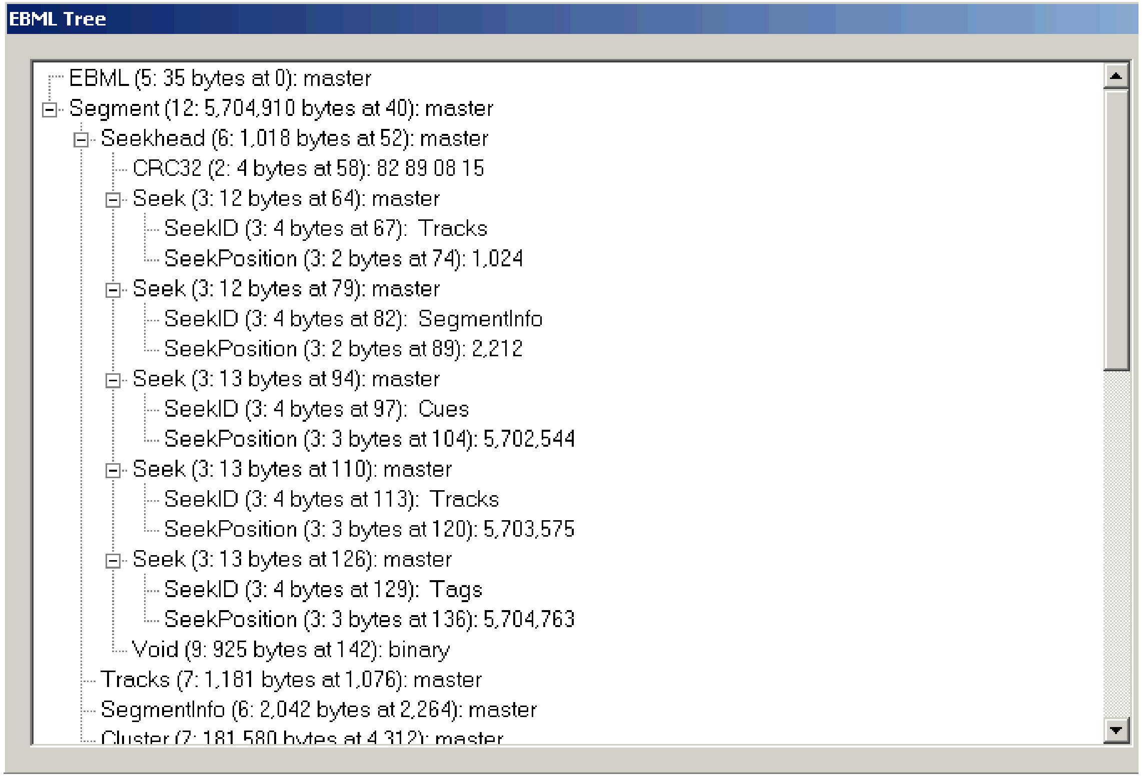
Table 5: The SEEK element, child of SEEKHEAD (→4)

Element	Description
uint, # $= 1$ **SEEKID** ID: `53 AB`	The **SEEKID** element contains the EBML-ID of the element found at the given position
uint, # $= 1$ **SEEKPOSITION** ID: `53 AC`	The **SEEKPOSITION** element contains the position relatively to the **SEGMENT**'s data at which an element with the ID **$SEEKID** can be found.

Index →page 2 *end of* **SEEK**

Not all Level 1 elements need to be included. Typical **SEEKHEADs** either include a list of all Level 1 elements, or a list of all Level 1 elements except for **CLUSTERS** (→ section 5.5). **SEEKHEADs** can also include references to other **SEEKHEADs** if there is, for example, a small **SEEKHEAD** at the beginning of the file and a larger one at its end.

The following picture illustrates the **SEEKHEAD** element in a real file. Note that the EBML Tree Viewer replaced Level 1 IDs in **SEEKID** with their human-readable name:



```
EBML (5: 35 bytes at 0): master
Segment (12: 5,704,910 bytes at 40): master
  Seekhead (6: 1,018 bytes at 52): master
    CRC32 (2: 4 bytes at 58): 82 89 08 15
    Seek (3: 12 bytes at 64): master
      SeekID (3: 4 bytes at 67):  Tracks
      SeekPosition (3: 2 bytes at 74): 1,024
    Seek (3: 12 bytes at 79): master
      SeekID (3: 4 bytes at 82):  SegmentInfo
      SeekPosition (3: 2 bytes at 89): 2,212
    Seek (3: 13 bytes at 94): master
      SeekID (3: 4 bytes at 97):  Cues
      SeekPosition (3: 3 bytes at 104): 5,702,544
    Seek (3: 13 bytes at 110): master
      SeekID (3: 4 bytes at 113):  Tracks
      SeekPosition (3: 3 bytes at 120): 5,703,575
    Seek (3: 13 bytes at 126): master
      SeekID (3: 4 bytes at 129):  Tags
      SeekPosition (3: 3 bytes at 136): 5,704,763
    Void (9: 925 bytes at 142): binary
  Tracks (7: 1,181 bytes at 1,076): master
  SegmentInfo (6: 2,042 bytes at 2,264): master
  Cluster (7: 181,580 bytes at 4,312): master
```

5.4 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.

Each track is described in one **TRACKENTRY**. Theoretically, using the **TRACKUID**, information about one track could be spread over different **TRACKENTRYS**, the UID would allow to know which track the information applies to, however, it is highly discouraged to stretch the specification like this.

Also, an empty **TRACKS** element would be rather useless, but should not lead to a parser error since the file can be played if all tracks are defined *somewhere*. Especially pure chapter files might have an empty **TRACKS** element if the muxer doesn't catch the case that no tracks are present and consequently creates an empty **TRACKS** element.

An example of a **TRACKENTRY** element can be found on (→ page 25)

Table 6: The TRACKS element, child of SEGMENT (→2)

Element	Description
Master, # ≥ 1 **TRACKENTRY** (→7) ID: `AE`	One **TRACKENTRY** element describes one track of the **SEGMENT**

Index →page 2

end of **TRACKS**

Table 7: The TRACKENTRY element, child of TRACKS (→6)

Element	Description
uint, # $= 1$ **(!)** **TRACKNUMBER** ID: `D7`	defines an identification number of the track. This number cannot be equal to 0. This number is used by the **BLOCK** and **SIMPLEBLOCK** structures.
uint, # $= 1$ **TRACKUID** ID: `73 C5`	is a unique identificator of the track within the file. It cannot be equal to 0

TRACKENTRY *continued on next page*

Element	Description
uint, # $= 1$ **(!)** **TRACKTYPE** (→13) ID: `83`	defines the type of a track, i.e. video, audio, subtitle etc.
bool, # ≤ 1 **FLAGENABLED** ID: `B9` def: 1	When **FLAGENABLED** is 1, track is used
bool, # ≤ 1 **FLAGDEFAULT** ID: `88` def: 1	When **FLAGDEFAULT** is 1, the track should be selected by the player by default. Obviously, if no video track and/or no audio track has a default flag, one video track and one audio track should be chosen by the player, whereas no subtitle should be enabled if no subtitle has a default flag.
bool, # ≤ 1 **FLAGFORCED** ID: `55 AA` def: 0	When **FLAGFORCED** is 1, the track must be played. When several subtitle tracks are forced, the one matching the audio language should be chose. An example would be a subtitle track that cannot be disabled, like the one you find on the german DVD “Eiskalte Engel” when you select english audio. Since this flag can only be used to apply a restriction on digital content, it must be qualified as Digital Restrictions Management.
bool, # ≤ 1 **FLAGLACING** ID: `9C` def: 0	When **FLAGLACING** is 1, the track may contain laced blocks. A parser that supports all types of lacing (→ section 6.2) can safely ignore this flag.
uint, # ≤ 1 **MINCACHE** ID: `6D E7` def: 0	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.
uint, # ≤ 1 **MAXCACHE** ID: `6D F8`	indicates the maximum cache size a player needs to cache frames. A value of NULL means that no cache is required.

TRACKENTRY *continued on next page*

Element	Description
uint, $\# \leq 1$ **DEFAULTDURATION** ID: `23 E3 83`	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 (→ section 6.1) is used. A value of 0 means that the duration of frames of the track is not necessarily constant (e.g. variable framerate video, or Vorbis audio). **DEFAULTDURATION** should be written for each track with a constant frame rate since it makes seeking easier.
float, $\# \leq 1$ **TRACKTIMECODESCALE** ID: `23 31 4F`	Every timecode of a block (`cluster timecode + block timecode`) is multiplied by this value to obtain the real timecode of a block.
utf-8, $\# \leq 1$ **NAME** ID: `53 6E`	A **NAME** element contains a human-readable name for the track. Note that you can't define which language this track name is in. You have to use Tags (→ section 5.9)) if you want to use several titles in different languages for the same track.
string, $\# \leq 1$ **LANGUAGE** ID: `22 B5 9C` def: eng	specifies the language of a track, using ISO639-2[3]. This is NOT necessarily the language of **$NAME**, for example a german AC3 track could be called "German - AC3 5.1" or "Deutsch - AC3 5.1" or "Allemand AC3 5.1" etc.
string, $\# = 1$ **(!)** **CODECID** ID: `86`	The **CODECID** specifies the Codec[4] which is used to decode the track.
binary, $\# \leq 1$ **CODECPRIVATE** ID: `63 A2`	**CODECPRIVATE** contains information the codec needs before decoding can be started. An example is the Vorbis initialization packets for Vorbis audio.
utf-8, $\# \leq 1$ **CODECNAME** ID: `25 86 88`	**CODECNAME** is a human-readable name of the Codec
uint, $\# \geq 0$ **ATTACHMENTLINK** ID: `74 46`	An **ATTACHMENTLINK** contains the UID of an attachment that is used by this track.

TRACKENTRY *continued on next page*

---

[3] `http://lcweb.loc.gov/standards/iso639-2/englangn.html`

[4] `http://matroska.org/technical/specs/codecid/index.html`

Element	Description
Master, # ≤ 1 **VIDEO** (→8) ID: `E0`	**VIDEO** contains information that is specific for video tracks
Master, # ≤ 1 **AUDIO** (→9) ID: `E1`	**AUDIO** contains information that is specific for audio tracks
Master, # ≤ 1 **CONTENTENCODINGS** (→10) ID: `6D 80`	**CONTENTENCODINGS** contains information about (lossless) compression or encryption of the track

Index →page 2

end of **TRACKENTRY**

Obviously, the **VIDEO** element must be present for video tracks, whereas the **AUDIO** element must be present for audio tracks. Although it doesn't make sense to have both elements in one **TRACKENTRY** element, it wouldn't make a file unplayable.

Table 8: The VIDEO element, child of TRACKENTRY (→7)

Element	Description
uint, # = 1 **PIXELWIDTH** ID: `B0`	Width of the encoded video track in pixels
uint, # ≤ 1 **PIXELHEIGHT** ID: `BA`	Height of the encoded video in pixels
uint, # ≤ 1 **PIXELCROPBOTTOM** ID: `54 AA` def: 0	Number of Pixels to be cropped from the bottom
uint, # ≤ 1 **PIXELCROPTOP** ID: `54 BB` def: 0	Number of Pixels to be cropped from the top

VIDEO *continued on next page*

Element	Description
uint, # ≤ 1 **PIXELCROPLEFT** ID: `54 CC` def: 0	Number of Pixels to be cropped from the left
uint, # ≤ 1 **PIXELCROPRIGHT** ID: `54 DD` def: 0	Number of Pixels to be cropped from the right
uint, # ≤ 1 **DISPLAYWIDTH** ID: `54 B0` def: **$PIXELWIDTH**	Width of the video during playback
uint, # ≤ 1 **DISPLAYHEIGHT** ID: `54 BA` def: **$PIXELHEIGHT**	Height of the video during playback
uint, # ≤ 1 **DISPLAYUNIT** ID: `54 B2` def: 0	Unit **$DISPLAYWIDTH** and **$DISPLAYHEIGHT** is measured in. This can be 0→pixels, 1→centimeters, 2→inches

Index →page 2

end of **VIDEO**

$PIXELCROPXXXX is applied on **$PIXELXXX**, so the output is cropped after decoding, but before stretching it to the dimensions indicated with **$DISPLAYXXXX**.

Table 9: The AUDIO element, child of TRACKENTRY (→7)

Element	Description
float, # ≤ 1 **SAMPLINGFREQUENCY** ID: `B5` def: 8 kHz	Indicates the sample rate the track is encoded at in Hz

AUDIO *continued on next page*

Element	Description
float, # ≤ 1 **OUTPUTSAMPLINGFREQUENCY** ID: `78 B5`	Indicates the sample rate the track must be played at in Hz. The default value of this element is equal to **$SAMPLINGFREQUENCY**.
uint, # ≤ 1 **CHANNELS** ID: `9F` def: 1	Number of channels of the audio track
uint, # ≤ 1 **BITDEPTH** ID: `62 64`	Bits per sample, this is usually used with PCM-Audio.

Index →page 2 *end of* **AUDIO**

Table 10: The CONTENTENCODINGS element, child of TRACKENTRY (→7)

Element	Description
Master, # ≥ 1 **CONTENTENCODING** (→11) ID: `62 40`	A **CONTENTENCODING**-element describes one compression or encryption that has been used on this track.

Index →page 2 *end of* **CONTENTENCODINGS**

Table 11: The CONTENTENCODING element, child of CONTENTENCODINGS (→10)

Element	Description
uint, # ≤ 1 **CONTENTENCODINGORDER** ID: `50 31` def: 0	Tells when to decode according to this pattern. The decoder starts with the **CONTENTENCODING** that has the highest **CONTENTENCODINGORDER**.

CONTENTENCODING *continued on next page*

Element	Description
uint, # ≤ 1 **CONTENTENCODING-SCOPE** (→14) ID: `50 32` def: 1	Defines which parts of the track are compressed or encrypted this way
uint, # ≤ 1 **CONTENTENCODING-TYPE** ID: `50 33` def: 0	Describes which type of encoding is described. 0 → compression, 1 → encryption
Master, # ≤ 1 **CONTENTCOMPRESSION** (→12) ID: `50 34`	If **CONTENTENCODINGTYPE=0**, this element describes how it is compressed
Master, # ≤ 1 **CONTENTENCRYPTION** (→**??**) ID: `50 35`	If **CONTENTENCRYPTION=1**, this element describes how it is encrypted

Index →page 2 *end of* **CONTENTENCODING**

The **CONTENTENCODING** element allows to apply not only encryption, but also lossless compression to a track. This can be used to compress text subtitles, but also to remove sync headers from audio packets. For example, each AC3 frame starts with `0B 77`, and there is no real point in saving those two bytes for each frame in a MATROSKA file. For a simple AC3 file, this does make sense because there it can be used to find a new frame start if data is damaged.

Table 12: The CONTENTCOMPRESSION element, child of CONTENTENCODING (→11)

Element	Description
uint, # ≤ 1 **CONTENTCOMPALGO** (→15) ID: `42 54` def: 0	The **CONTENTCOMPALGO** element says which algorithm was used for this compression.

CONTENTCOMPRESSION *continued on next page*

Element	Description
binary, # ≤ 1 **CONTENTCOMPSETTINGS** ID: `42 55`	Contains settings that are required for decompression. These settings are specific for each compression algorithm. For example, it contains the striped header bytes when **$CONTENTCOMPALGO**=3 (→ page 25).

Index →page 2

end of **CONTENTCOMPRESSION**

Table 13: Values of TRACKTYPE, child of TRACKENTRY (→7)	
Value	Description
0x01	track is a video track
0x02	track is an audio track
0x03	track is a complex track, i.e. a combined video and audio track
0x10	track is a logo track
0x11	track is a subtitle track
0x12	track is a button track
0x20	track is a control track

end of **TRACKTYPE**

Table 14: Bits in CONTENTENCODINGSCOPE, child of CONTENTENCODING (→11)	
Value	Description
1	all frames
2	the track's **CODECPRIVATE**
4	the **CONTENTCOMPRESSION** in the next **CONTENTENCODING** (next as in next in decoding order)

end of **CONTENTENCODINGSCOPE**

Here is one example of a possible **TRACKENTRY** element: A DTS-audio track that is using header striping. The **CONTENTCOMPSETTINGS** element contains the four bytes each DTS frame starts with.

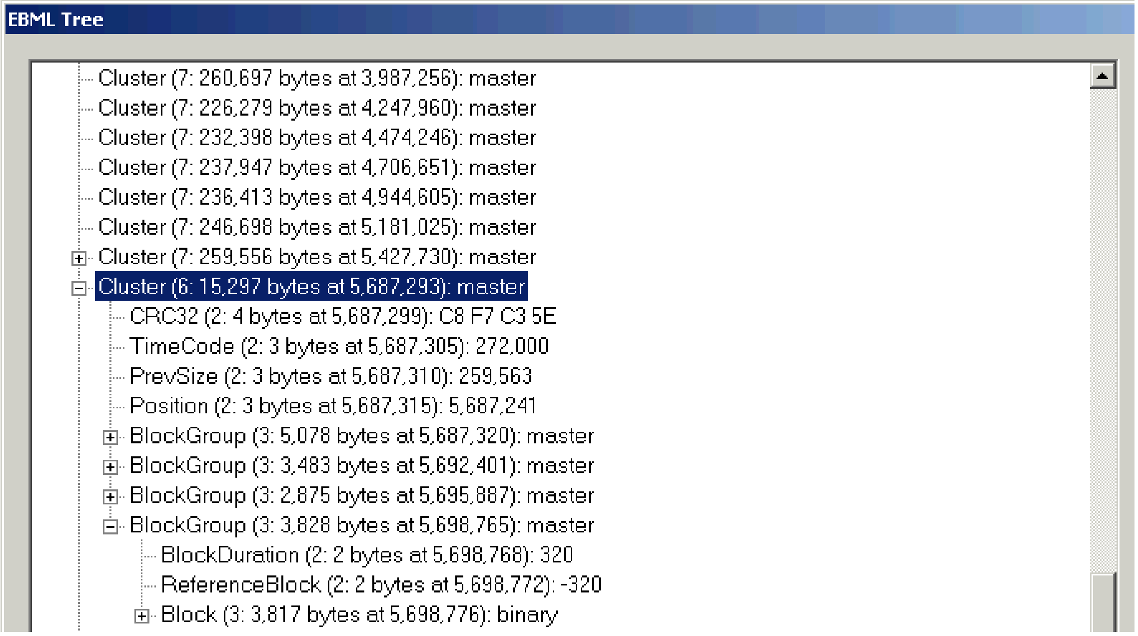
Table 15: Values of CONTENTCOMPALGO, child of CONTENTCOMPRESSION (→12)

Value	Description
0	zlib
1	bzlib
2	lzo1x
3	header striping

end of **CONTENTCOMPALGO**

5.5 Cluster

A **CLUSTER** contains multimedia data and usually spans over a range of a few seconds. The following picture shows a typical cluster:



```
  Cluster (7: 260,697 bytes at 3,987,256): master
  Cluster (7: 226,279 bytes at 4,247,960): master
  Cluster (7: 232,398 bytes at 4,474,246): master
  Cluster (7: 237,947 bytes at 4,706,651): master
  Cluster (7: 236,413 bytes at 4,944,605): master
  Cluster (7: 246,698 bytes at 5,181,025): master
⊞ Cluster (7: 259,556 bytes at 5,427,730): master
⊟ Cluster (6: 15,297 bytes at 5,687,293): master
    CRC32 (2: 4 bytes at 5,687,299): C8 F7 C3 5E
    TimeCode (2: 3 bytes at 5,687,305): 272,000
    PrevSize (2: 3 bytes at 5,687,310): 259,563
    Position (2: 3 bytes at 5,687,315): 5,687,241
  ⊞ BlockGroup (3: 5,078 bytes at 5,687,320): master
  ⊞ BlockGroup (3: 3,483 bytes at 5,692,401): master
  ⊞ BlockGroup (3: 2,875 bytes at 5,695,887): master
  ⊟ BlockGroup (3: 3,828 bytes at 5,698,765): master
      BlockDuration (2: 2 bytes at 5,698,768): 320
      ReferenceBlock (2: 2 bytes at 5,698,772): -320
    ⊞ Block (3: 3,817 bytes at 5,698,776): binary
```

Although sticking to this order of the elements is not mandatory, it is recommended not to have any non-**BLOCKGROUP/SIMPLEBLOCK** after the first **BLOCKGROUP/SIMPLEBLOCK**, because it's bad if the entire cluster must be read before it can be used just because the timecode is stored at the end.

Table 16: The CLUSTER element, child of SEGMENT (→2)

Element	Description
uint, # ≤ 1 **TIMECODE** ID: `E7` def: 0	The Cluster timecode is the timecode all block timecodes are indicated relatively to.
uint, # ≤ 1 **POSITION** ID: `A7`	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.

CLUSTER *continued on next page*

Element	Description
uint, # ≤ 1 **PrevSize** ID: `AB`	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.
Master, # ≥ 0 **BlockGroup** (→17) ID: `A0`	Contains a **Block** along with some attached information like references
binary, # ≥ 0 **SimpleBlock** ID: `A3`	This is a **Block** (→ page 39) without additional attached information. Since a **SimpleBlock** does not require a **BlockGroup** around it, it causes less overhead. **SimpleBlock** is Matroska v2.

Index →page 2 *end of* **Cluster**

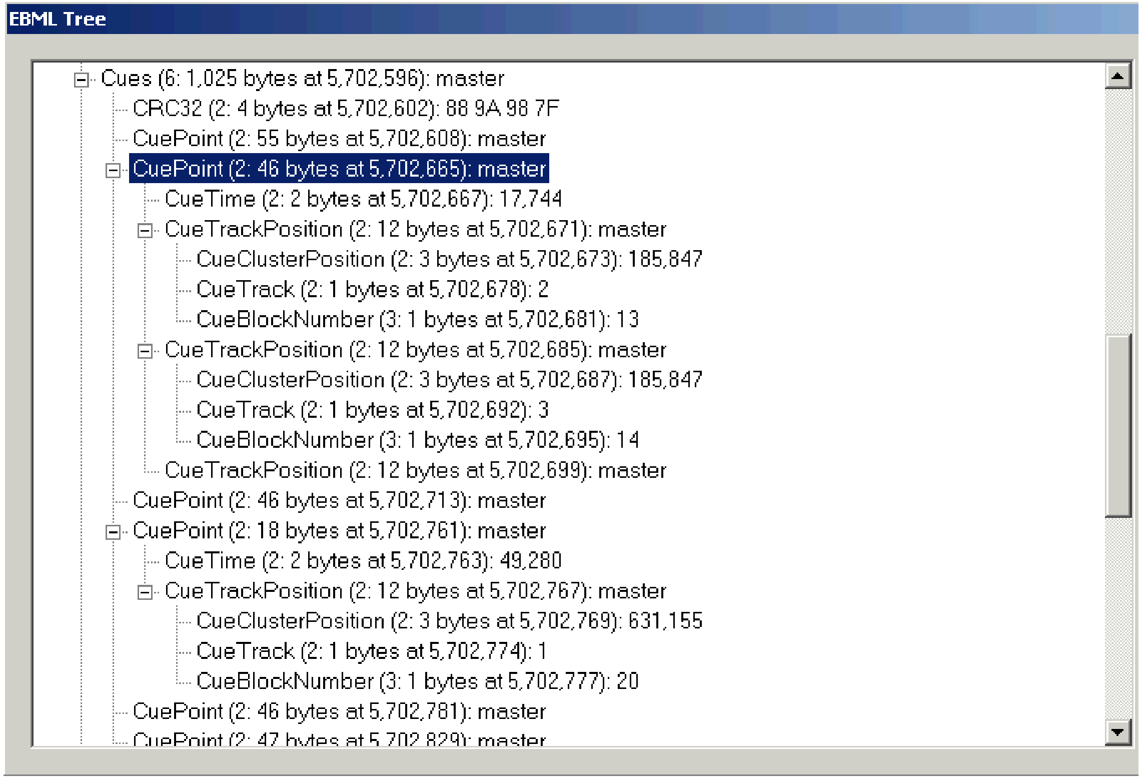
Table 17: The BlockGroup element, child of Cluster (→16)

Element	Description
binary, # $= 1$ (!) **Block** ID: `A1`	contains data to be replayed. See page 39 for details.
int, # ≥ 0 **ReferenceBlock** ID: `FB`	Timecode of a frame, relative to the **Block**'s timecode, of a frame that needs to be decoded before this **Block** can be decoded.
int, # ≤ 1 **BlockDuration** ID: `9B`	Indicates the scaled duration of the **block**. If this value is not written, it is assumed to be (1) the difference `<timecode of next block of the same stream> - <timecode>` (2) equal to **DefaultDuration** (for the last block of each stream). As a consequence, the **Duration** element is mandatory for every **block** of subtitle tracks, unless a subtitle is indeed supposed to disappear only directly before the next one appears. But even then it is recommended to write **Duration**.

Index →page 2 *end of* **BlockGroup**

5.6 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.



```
⊟ Cues (6: 1,025 bytes at 5,702,596): master
    CRC32 (2: 4 bytes at 5,702,602): 88 9A 98 7F
    CuePoint (2: 55 bytes at 5,702,608): master
  ⊟ CuePoint (2: 46 bytes at 5,702,665): master
      CueTime (2: 2 bytes at 5,702,667): 17,744
    ⊟ CueTrackPosition (2: 12 bytes at 5,702,671): master
        CueClusterPosition (2: 3 bytes at 5,702,673): 185,847
        CueTrack (2: 1 bytes at 5,702,678): 2
        CueBlockNumber (3: 1 bytes at 5,702,681): 13
    ⊟ CueTrackPosition (2: 12 bytes at 5,702,685): master
        CueClusterPosition (2: 3 bytes at 5,702,687): 185,847
        CueTrack (2: 1 bytes at 5,702,692): 3
        CueBlockNumber (3: 1 bytes at 5,702,695): 14
      CueTrackPosition (2: 12 bytes at 5,702,699): master
    CuePoint (2: 46 bytes at 5,702,713): master
  ⊟ CuePoint (2: 18 bytes at 5,702,761): master
      CueTime (2: 2 bytes at 5,702,763): 49,280
    ⊟ CueTrackPosition (2: 12 bytes at 5,702,767): master
        CueClusterPosition (2: 3 bytes at 5,702,769): 631,155
        CueTrack (2: 1 bytes at 5,702,774): 1
        CueBlockNumber (3: 1 bytes at 5,702,777): 20
    CuePoint (2: 46 bytes at 5,702,781): master
    CuePoint (2: 47 bytes at 5,702,829): master
```

Table 18: The CUES element, child of SEGMENT (→2)	
Element	Description
Master, # ≥ 1 **CUEPOINT** (→19) ID: `BB`	One **CUEPOINT** contains one entry point (or a list of entry points with one point for one track) for one timecode.
Index →page 2	*end of* **CUES**

Table 19: The CUEPOINT element, child of CUES (→18)

Element	Description
uint, # $= 1$ (!) **CUETIME** ID: `B3`	The timecode of the **CLUSTER**s or **BLOCK**s that are referred to by this **CUEPOINT**
Master, # ≥ 1 **CUETRACKPOSITIONS** (→20) ID: `B7`	A position where a **CLUSTER** or **BLOCK** can be found with the timecode **$CUETIME**.

Index →page 2

end of **CUEPOINT**

Table 20: The CUETRACKPOSITIONS element, child of CUEPOINT (→19)

Element	Description
uint, # ≥ 1 (!) **CUETRACK** ID: `F7`	Track for which a position is given. This track number is the same as **TRACKENTRY** (→ Table 7)::**TRACKNUMBER**.
uint, # ≥ 1 (!) **CUECLUSTERPOSITION** ID: `F1`	The position of the cluster the referred block is found in. This position is relative to the **SEGMENT**'s (→ Table 2) data section.
uint, # ≤ 1 **CUEBLOCKNUMBER** ID: `53 78`	The block with timecode **$CUETIME** is the **$CUEBLOCKNUMBER**-th **BLOCK/SIMPLEBLOCK** inside the **CLUSTER** at position **$CUECLUSTERPOSITION**.

Index →page 2

end of **CUETRACKPOSITIONS**

5.7 Chapters - Editions and ChapterAtoms

The **CHAPTERS** element contains a list of all editions and chapters found in this **SEGMENT**. Chapters in MATROSKA files are more powerful than chapters on DVDs, their handling is, however, way more complex.

Table 21: The CHAPTERS element, child of SEGMENT (→2)

Element	Description
Master, # ≥ 1 **EDITIONENTRY** (→22) ID: `45 B9`	One **EDITIONENTRY** describes one Edition. Just like with **TRACKENTRY** (→ Table 7), theoretically you could spread information about one Edition over different **EDITIONENTRYS** and use **$EDITIONUID** to find out which edition the **EDITIONENTRY** is referring to, but it's highly discouraged.

Index →page 2 *end of* **CHAPTERS**

An edition contains one set of chapter definitions, so having several editions means having several sets of chapter definitions. This case is used when using this as a playlist - playing one chapter after the other while having gaps between the chapters.

Table 22: The EDITIONENTRY element, child of CHAPTERS (→21)

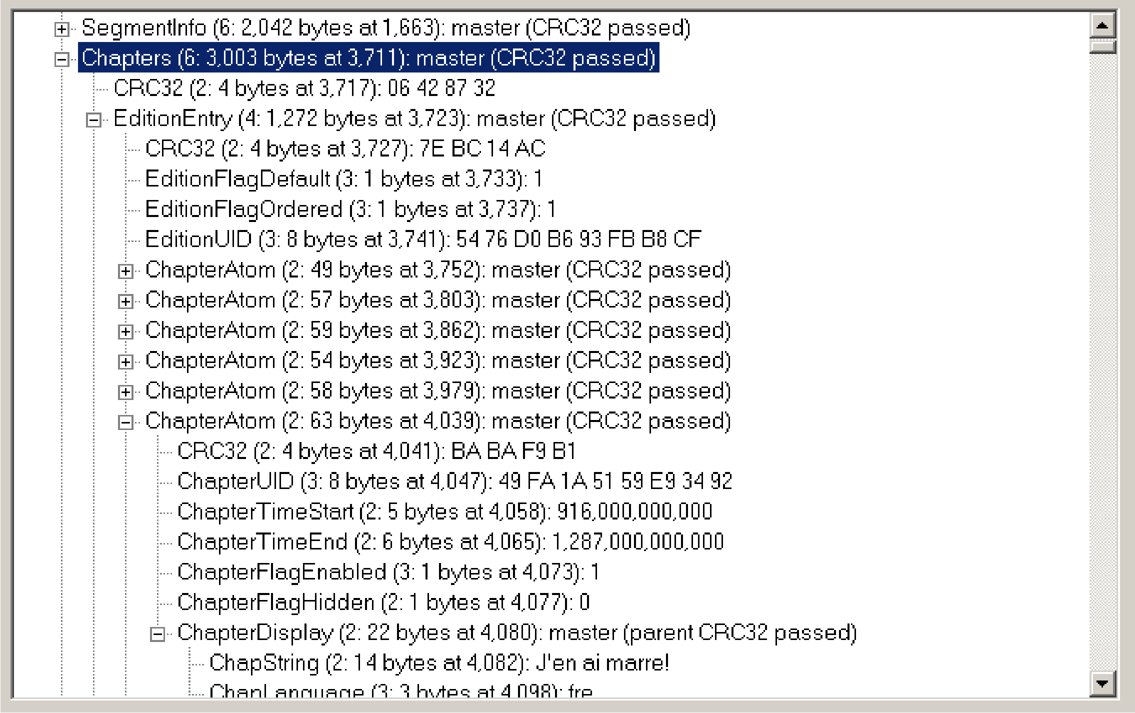
Element	Description
uint, # ≤ 1 **EDITIONUID** ID: `45 BC`	**$EDITIONUID** is the UID of the edition. This element is mandatory if you want to apply one or more titles to an edition
bool, # ≤ 1 **EDITIONFLAGHIDDEN** ID: `45 BD` def: 0	When **$EDITIONFLAGHIDDEN** is 1, this edition should not be available via the user interface
bool, # ≤ 1 **EDITIONFLAGDEFAULT** ID: `45 DB` def: 0	When **$EDITIONFLAGDEFAULT** is 1, this edition should be selected by the player as default

EDITIONENTRY *continued on next page*

Element	Description
bool, # ≤ 1 **EDITIONFLAGORDERED** ID: `45 DD` def: 0	When **$EDITIONFLAGORDERED** is 1, this edition contains a playlist. When **$EDITIONFLAGORDERED** is 0, it contains a simple DVD like chapter definition.
Master, # ≥ 1 **CHAPTERATOM** (→23) ID: `B6`	One **CHAPTERATOM** contains the definition of one chapter. This element is the only one in MATROSKA files that can contain itself recursively - in this case to define subchapters.

Index →page 2 *end of* **EDITIONENTRY**

The following picture shows an ordered edition:



```
⊞ SegmentInfo (6: 2,042 bytes at 1,663): master (CRC32 passed)
⊟ Chapters (6: 3,003 bytes at 3,711): master (CRC32 passed)
   CRC32 (2: 4 bytes at 3,717): 06 42 87 32
  ⊟ EditionEntry (4: 1,272 bytes at 3,723): master (CRC32 passed)
     CRC32 (2: 4 bytes at 3,727): 7E BC 14 AC
     EditionFlagDefault (3: 1 bytes at 3,733): 1
     EditionFlagOrdered (3: 1 bytes at 3,737): 1
     EditionUID (3: 8 bytes at 3,741): 54 76 D0 B6 93 FB B8 CF
    ⊞ ChapterAtom (2: 49 bytes at 3,752): master (CRC32 passed)
    ⊞ ChapterAtom (2: 57 bytes at 3,803): master (CRC32 passed)
    ⊞ ChapterAtom (2: 59 bytes at 3,862): master (CRC32 passed)
    ⊞ ChapterAtom (2: 54 bytes at 3,923): master (CRC32 passed)
    ⊞ ChapterAtom (2: 58 bytes at 3,979): master (CRC32 passed)
    ⊟ ChapterAtom (2: 63 bytes at 4,039): master (CRC32 passed)
       CRC32 (2: 4 bytes at 4,041): BA BA F9 B1
       ChapterUID (3: 8 bytes at 4,047): 49 FA 1A 51 59 E9 34 92
       ChapterTimeStart (2: 5 bytes at 4,058): 916,000,000,000
       ChapterTimeEnd (2: 6 bytes at 4,065): 1,287,000,000,000
       ChapterFlagEnabled (3: 1 bytes at 4,073): 1
       ChapterFlagHidden (2: 1 bytes at 4,077): 0
      ⊟ ChapterDisplay (2: 22 bytes at 4,080): master (parent CRC32 passed)
         ChapString (2: 14 bytes at 4,082): J'en ai marre!
         ChapLanguage (3: 3 bytes at 4,098): fre
```

Table 23: The CHAPTERATOM element, child of EDITIONENTRY (→22), child of CHAPTERATOM (→23)	
Element	Description
uint, # = 1 **CHAPTERUID** ID: `73 C4`	The UID of this chapter. It must be unique within the file.

CHAPTERATOM *continued on next page*

Element	Description
uint, # ≤ 1 **ChapterTimeStart** ID: `91` def: 0	The *unscaled* timecode the chapter starts at. As the value is unsigned, a chapter cannot start earlier than at timecode 0, even whereas timecodes up to -30.000 are possible for multimedia data.
uint, # ≤ 1 **ChapterTimeEnd** ID: `92`	The *unscaled* timecode the chapter ends at. The default value is the start of the next chapter or the end of the parent chapter or the end of the segment, whatever exists, in that order.
bool, # ≤ 1 **ChapterFlagHidden** ID: `98` def: 0	When **$ChapterFlagHidden** is 1, the chapter should not be visible in the user interface, but should be played back normally.
bool, # ≤ 1 **ChapterFlagEnabled** ID: `45 98` def: 1	When **$ChapterFlagEnabled** is 0, the chapter should be skipped by the player
char[16], # ≤ 1 **ChapterSegmentUID** ID: `6E 67`	This element can only occur if **$EditionFlagOrdered**=1. The **Segment** of which the UID is **$ChapterSegmentUID** is used instead of the current **Segment**. Obviously, this **Segment** should be easy to find, like when it is the first segment of a file in the same directory.
uint, # ≤ 1 **ChapterSegment** **-EditionUID** ID: `6E BC`	The edition to use inside the **Segment** selected via **ChapterSegmentUID**. The timecodes **$ChapterTimeStart** and **$ChapterTimeEnd** refer to playback timecodes of that edition, i.e. the timecodes are relative to that playlist. This is called “nested Editions” and is NOT SUPPORTED by *Haali Media Splitter*.
Master, # ≤ 1 **ChapterTracks** (→24) ID: `8F`	Contains a list of tracks the chapter applies to.
Master, # ≥ 0 **ChapterDisplay** (→25) ID: `80`	Contains all chapter titles

Index →page 2

end of **ChapterAtom**

A useful application for the **CHAPTERFLAGHIDDEN** element in connection with ordered editions is the following: You have a couple of episodes of a series, but want to save space by only saving the intro and outtro once. You create one playlist (ordered edition) per episode, and another playlist playing all episodes in a row. Whereas in the first case you might want to play intro and outtro for each episode, you might not want to do that in the second case.

If you don't want to make the three parts intro - movie - outtro selectable via the user interface when playing single episodes, you call the intro-chapter "Episode - blah" and hide the movie- and the outtro chapter using **$CHAPTERFLAGHIDDEN**=1. Then, the playlist playing all episodes would be *intro - episode 1 - episode 2 - ... - last episode - outtro*, whereas the other playlists would be *intro - episode* N - *outtro*. The name of the intro chapter would be set to "Episode *n*".

Table 24: The CHAPTERTRACKS element, child of CHAPTERATOM (→23)

Element	Description
uint, # ≥ 1 **CHAPTERTRACKNUMBER** ID: `89`	One number of a track a chapter is used with.

Index →page 2 *end of* **CHAPTERTRACKS**

Table 25: The CHAPTERDISPLAY element, child of CHAPTERATOM (→23)

Element	Description
utf-8, # ≤ 1 **CHAPSTRING** ID: `85`	A title of a chapter
string, # ≥ 0 **CHAPLANGUAGE** ID: `43 7C` def: eng	The language of **$CHAPSTRING** as defined in ISO639-2[5]
utf-8, # ≥ 0 **CHAPCOUNTRY** ID: `43 7E`	A country the title is used in. For example, a german title in Germany might be different than the title used in Austria.

Index →page 2 *end of* **CHAPTERDISPLAY**

---

[5] `http://lcweb.loc.gov/standards/iso639-2/englangn.html#two`

5.8 Attachments

Theoretically, any file type can be attached to a MATROSKA file, however, this possibility is usually used to attach pictures like CD covers or fonts required to display a subtitle track correctly. Obviously, attaching executable files would allow for MATROSKA files to contain viruses - a scenario that is not exactly the indended application of attachments or anything else MATROSKA is capable of.

Table 26: The ATTACHMENTS element, child of SEGMENT (→2)

Element	Description
Master, # ≥ 1 **ATTACHEDFILE** (→27) ID: `61 A7`	Describes and contains one attached file

Index →page 2 *end of* **ATTACHMENTS**

Table 27: The ATTACHEDFILE element, child of ATTACHMENTS (→26)

Element	Description
utf8, # ≤ 1 **FILEDESCRIPTION** ID: `46 7E`	A human-readable description of the file
utf8, # ≤ 1 **FILENAME** ID: `46 6E`	The name that should be proposed by a demuxer when extracting the file
string, # ≤ 1 **FILEMIMETYPE** ID: `46 60`	MIME type of the file, like ...
binary, # ≤ 1 **FILEDATA** ID: `46 5C`	The file itself
uint, # $= 1$ **FILEUID** ID: `46 AE`	The UID of that file, just like **TRACKUID**, **CHAPTERUID** etc. The UID is required if a **TRACKENTRY** (→ Table 7) wants to refer to this Attachment.

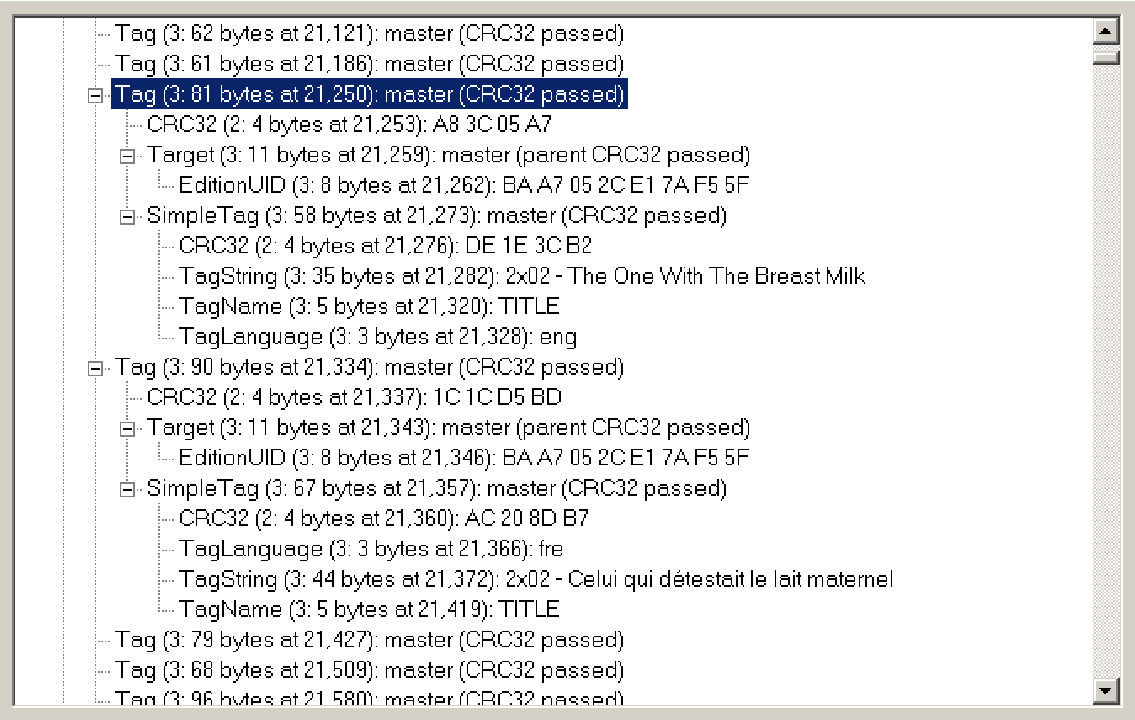
Index →page 2 *end of* **ATTACHEDFILE**

5.9 Tags

Table 28: The TAGS element, child of SEGMENT (→2)

Element	Description
Master, # ≥ 1 **TAG** (→29) ID: `73 73`	One **TAG** element describes one Tag
Index →page 2	*end of* **TAGS**

TAGS provide additional information[6] not important for replay. A **TAGS** element contains a number of **TAG** elements. Each **TAG** element contains a list of UIDs (usually **TRACKUID**s or **EDITIONUID**s), and a list of **SIMPLETAG**s, each one containing a name and a value:



```
Tag (3: 62 bytes at 21,121): master (CRC32 passed)
Tag (3: 61 bytes at 21,186): master (CRC32 passed)
Tag (3: 81 bytes at 21,250): master (CRC32 passed)
  CRC32 (2: 4 bytes at 21,253): A8 3C 05 A7
  Target (3: 11 bytes at 21,259): master (parent CRC32 passed)
    EditionUID (3: 8 bytes at 21,262): BA A7 05 2C E1 7A F5 5F
  SimpleTag (3: 58 bytes at 21,273): master (CRC32 passed)
    CRC32 (2: 4 bytes at 21,276): DE 1E 3C B2
    TagString (3: 35 bytes at 21,282): 2x02 - The One With The Breast Milk
    TagName (3: 5 bytes at 21,320): TITLE
    TagLanguage (3: 3 bytes at 21,328): eng
Tag (3: 90 bytes at 21,334): master (CRC32 passed)
  CRC32 (2: 4 bytes at 21,337): 1C 1C D5 BD
  Target (3: 11 bytes at 21,343): master (parent CRC32 passed)
    EditionUID (3: 8 bytes at 21,346): BA A7 05 2C E1 7A F5 5F
  SimpleTag (3: 67 bytes at 21,357): master (CRC32 passed)
    CRC32 (2: 4 bytes at 21,360): AC 20 8D B7
    TagLanguage (3: 3 bytes at 21,366): fre
    TagString (3: 44 bytes at 21,372): 2x02 - Celui qui détestait le lait maternel
    TagName (3: 5 bytes at 21,419): TITLE
Tag (3: 79 bytes at 21,427): master (CRC32 passed)
Tag (3: 68 bytes at 21,509): master (CRC32 passed)
Tag (3: 96 bytes at 21,580): master (CRC32 passed)
```

If no **TARGETS** are specified, then the **TAG** is a global **TAG** refering to the entire **SEGMENT**. Of course, two different **TAG** elements can contain identical **TARGETS**.

---

[6] `http://www.matroska.org/technical/specs/tagging/index.html`

Table 29: The TAG element, child of TAGS (→28)	
Element	Description
Master, # ≤ 1 **TARGETS** (→30) ID: `63 C0`	Describes which elements a Tag applies to
Master, # ≥ 1 **SIMPLETAG** (→31) ID: `67 C8`	Each **SIMPLETAG** contains one tag that applies to each target in **TARGETS**

Index →page 2 *end of* **TAG**

Note that there is nothing like a **TAGUID**.

Table 30: The TARGETS element, child of TAG (→29)	
Element	Description
uint, # ≤ 1 **TARGETTYPEVALUE** (→**??**) ID: `68 CA` def: 50	This number describes the logical level of the object the Tag refers to
utf-8, # ≤ 1 **TARGETTYPE** ID: `63 CA`	A string describing the logical level of the object the Tag is refering to
uint, # ≥ 0 **TRACKUID** ID: `63 C5`	The UID of a track the tag is referring to
uint, # ≥ 0 **EDITIONUID** ID: `63 C9`	The UID of an edition the tag is referring to. Note that this is the only way to apply titles to an edition
uint, # ≥ 0 **CHAPTERUID** ID: `63 C4`	The UID of a chapter the tag is referring to
uint, # ≥ 0 **ATTACHMENTUID** ID: `63 C6`	The UID of an attachment the tag is referring to

Index →page 2 *end of* **TARGETS**

Table 31: The SIMPLETAG element, child of TAG (→29)	
Element	Description
utf-8, # ≥ 1 **(!)** **TAGNAME** ID: `45 A3`	Name of the tag.
string, # ≤ 1 **TAGLANGUAGE** ID: `44 7A` def: und	**$TAGLANGUAGE** is the language of **$TAGNAME**. Note that the default here is 'und', whereas the default track / chapter title language is 'eng'.
bool, # ≤ 1 **TAGORIGINAL** ID: `44 84` def: 1	When 1, this title and language is the original title given to the item
utf-8, # ≤ 1 **TAGSTRING** ID: `44 87`	The value of the tag when it is a string
binary, # ≤ 1 **TAGBINARY** ID: `44 85`	The 'value' of the tag when it's a binary tag

Index →page 2 *end of* **SIMPLETAG**

5.9.1 A few common Tags

- **TITLE**, Target: EditionUID: used to define names for Editions. This is exactly what you can see in the screenshot above.
- **BPS**, Target: TrackUID: used to define the bitrate of a track
- **FPS**, Target: TrackUID: used to define the framerate of a track

6 MATROSKA block Layout and Lacing

6.1 Basic layout of a Block

A MATROSKA block has the following format:

```
BLOCK {
  vint TrackNumber
  sint16 Timecode  // relative to Cluster timecode
  int8 Flags      // lacing, keyframe, discardable
  if (lacing) {
    int8 frame_count-1
    if (lacing == EBML lacing) {
       vint size[0]
       svint 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  0x80: keyframe:
              No frame after this frame can reference any frame before
              this frame and vice versa (in AVC-words: this frame is an
              IDR frame). The frame itself doesn't reference any other
              frames.
  Bits 0x06: lace type
              00 - no lacing
              01 - Xiph lacing
              11 - EBML lacing
              10 - fixed-size lacing
  Bit 0x08 : invisible: duration of this block is 0
  Bit 0x01 : discardable: this frame can be discarded if the decoder
             is slow
```

The following flags are only defined for Matroska v2 and can thus only be used in a **SIMPLEBLOCK**: *keyframe, invisible, discardable*. 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. This section assumes that **BLOCKGROUPS** and **BLOCKS** are used, and that no **SIMPLEBLOCKS** are used. If you want to estimate overhead of files that use **SIMPLEBLOCKS**, you get about the same overhead as with **BLOCKS** without **BLOCKDURATION**, **REFERENCEBLOCK** or **BLOCKGROUP**.

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**

BLOCKGROUPs 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 usually 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`[7] (short for 'matroska lace sim-

---

[7] `http://www-user.tu-chemnitz.de/~noe/Video-Zeug/mls/`

ulator'). 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								
Audio Format	4	8	12	16	24	32	48	64	96
MP3 @ 128 kbps	1,39	1,29	1,26	1,24	1,22	1,22	1,21	1,20	1,20
MP3 @ 192 kbps	1,50	1,41	1,38	1,37	1,36	1,35	1,34	1,34	1,33
HE-AAC @ 224 kbps	1,39	1,29	1,25	1,24	1,22	1,21	1,20	1,20	1,20
HE-AAC @ 64 kbps	1,34	1,23	1,19	1,18	1,16	1,15	1,14	1,14	1,13
LC-AAC @ 268 kbps	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 fewer **BLOCK**s and thus fewer **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								
Audio Format	4	8	12	16	24	32	48	64	96
MP3 @ 128 kbps	4,14	2,67	2,17	1,93	1,68	1,56	1,48	1,41	1,33
MP3 @ 192 kbps	4,25	2,79	2,30	2,06	1,81	1,75	1,61	1,54	1,47
HE-AAC @ 224 kbps	4,14	2,66	2,23	2,05	1,76	1,62	1,48	1,40	1,33
HE-AAC @ 64 kbps	4,09	2,61	2,11	1,86	1,62	1,49	1,40	1,34	1,27
LC-AAC @ 268 kbps	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. Nevertheless, 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 kbps	782kB	492kB	389kB
MP3 @ 192 kbps	817kB	530kB	430kB
HE-AAC @ 224 kbps	402kB	266kB	201kB
HE-AAC @ 64 kbps	395kB	245kB	192kB
LC-AAC @ 268 kbps	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 **BLOCKGROUPS**, 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 **CUEPOINTs** for audio tracks. In that case, as every frame will be a keyframe, the number of **CUEPOINTs** only depends on the muxing application. Predicting the overhead requires to know its behaviour.

8 Links

Matroska pages / software:

`http://www.matroska.org`
`http://haali.cs.msu.ru/mkv/`
`http://www.alexander-noe.com/`
`http://de.wikipedia.org/wiki/Matroska`
`http://www.matroska.info/`
`http://ld-anime.faireal.net/guide/jargon.matroska-en`