

School of Electronic  
Engineering and  
Computer Science

MSc Digital Music  
Processing

Final Report

**Development of**  
**an IM AF**  
**encoder**

Eugenio Oñate

Date: 29 of August, 2012

## Disclaimer

This report, with any accompanying documentation and/or implementation, is submitted as part requirement for the degree of MSc Digital Music Processing at the University of London.

It is the product of my own labour except where indicated in the text.  
The report may be freely copied and distributed provided the source is acknowledged.

## **Acknowledgment**

This thesis would not have been possible without the support of many people. The author wishes to express his gratitude to his supervisor, Prof. Panos Kudumakis who was abundantly helpful and offered invaluable assistance, support and guidance.

Deepest gratitude are also due to the members of the Centre of Digital Music of Queen Mary, Prof. George Fazekas and Stuart Mansbridge without whose knowledge and assistance this thesis would not have been successful.

The author wish to express his gratitude to Montserrat Pla and his beloved family; for their understanding, support and encourage through the duration of his studies.

## **Abstract**

In this thesis an encoder able to create a new interactive file format has been developed. For this purpose a program in C has been written following the standard defined by MPEG. This file is called Interactive Music Application Format (IM AF). The IM AF file format allows users more control over the song. More specifically, it permits to vary the volume of each instrument or choose a predefined preset. All the actions are supervised by rules, thereby protecting the initial intention of the author. In order to place the reader in context, the thesis describes the previous file formats, which have been the basis for the IM AF. Moreover, we compare the previous techniques that used the same idea of multi tracks formats, but failed with the interoperability between different players. Then we describe the key features of the new encoder and the steps taken in its development and implementation. The performance of the encoder has been validated by creating three IM AF files with special features. These conformance files have been successfully tested in the IM AF player provided by the MPEG group.

# Content

1. Introduction .....	1
1.1. Thesis statement .....	1
1.2. Motivation .....	1
1.3. Goals/Objectives .....	2
1.4. Contributions .....	3
2. Background research and literature review .....	5
2.1. Overview of MPEG .....	5
2.2. MPEG Audio Compression .....	6
2.3. MPEG-1 Layer-III .....	7
2.4. MPEG-4 .....	9
2.5. Previous interactive systems .....	10
3. Development of the IM AF file .....	14
3.1. Interactive music service .....	14
3.2. Creating an IM AF file .....	15
3.3. File structure of IM AF .....	17
4. Implementation .....	27
4.1. Design: How it works .....	27
4.2. Implementation: Explanation of the code .....	29
4.2.1 Header: IM AF Encoder .....	30
4.2.2 Main programme .....	30
4.3. Programs that analyse files .....	32
5. Results and evaluation .....	35
5.1. Conformance files .....	37
5.2. Advantages .....	39
5.3. Weaknesses .....	40
6. Conclusions and further work .....	41
References .....	43
Appendices.1 – Paper: Development of an IM AF Encoder .....	45
Appendices.2 – Listening of the IM AF encoder program .....	51
Main.c .....	51
IM AF Encoder.h .....	66

# **1. Introduction**

## **1.1 Thesis Statement**

This thesis will explain the process to create an encoder for the Interactive Music Application Format. This encoder allows more control over the song. Users can change the volume of each instrument or select predefined mixing presets.

The created file will be reproduced by the IM AF player to prove its compliance.

## **1.2 Motivation**

The music market is strangled with decreasing sales. Customers need new products that attract their attention. Now it is a very exciting moment to innovate and create a new file format that will change the way people listen the music. Formats like mp3 were a good revulsive for industry, but have been here since the 90s; today it is all about interactivity between user and technology. From the first recordings in Vinyl's through the Cassettes era and later with the CDs, customers have not had the opportunity to modify the songs and adapt them as desired. When a song arrives at the store it has a specific sound created by the musicians and the producer. Sometimes users have felt the curiosity to listen in detail an instrument of a song, or have wanted to turn down the volume of another instrument because it was annoying.

At present, technologies have led people to live in a society where everyone is connected among themselves and sharing all kind of information. The music industry has been reluctant to let people interact with the songs, basically because the technology was not ready. But today, with the expansion of smart phones and tablets, a great amount of people joining social networks and thousand of webpages linking songs from popular and unknown artists, it is the perfect opportunity to change the concept of the listener, making him/her participate of the experience. A new interactive music services has emerged. Future will be dominated by this new concept of digital music content; thus, is essential to have a standardized file format to ensure interoperability between different types of interactive services as music players and music albums.

The Moving Picture Experts Group (MPEG) defined a new file format called Interactive Music Application Format (IM AF) [1] as part of the Multimedia application format (MPEG-A) [2], which is a versatile file format standard for mixing different types of multimedia data (music, images and text). It allows users to modify the volume of each instrument separately or change the mixing style according to some presets predefined by the producer. Also, it

permits interoperability between interactive music contents: users can share different versions of the song (changing the mix) or add/ change instruments. Besides that, it supports metadata information as synchronized text (karaoke) and images (album cover).

This new file format opens a new range of possibilities for the music sector. Webpages like Myspace.com or Last.fm, where thousand of bands share their tracks, could use the IM AF file format as their standard instead of the regular MP3. This will have a direct impact on its success, as services interoperability is ensured. Webpages will not be restricted to just present the songs; instead they will provide a range of additional options that will provide a better experience to the user.

### **1.3 Goals/Objectives**

The creation process for standardized files like MPEG-1 Layer III (.mp3) or IM AF (.ima) is split in two sections: encoder and decoder. In the encoder section, the file is created following the appropriate standard; parts like headers, track information and samples data are put together inside a binary file. On the other hand, the decoder is responsible to read and understand the file so that it will be able to reproduce it.

When a new file is defined, as it is the case, the decoder of the file is made publicly available through ISO/IEC MPEG to let companies design their own encoder. Thereby, they ensure compatibility of the file no matter how it was created. Since MPEG defined the standard for Interactive Music Application Format in 2010 no one has publicly released an implementation of the encoder although commercial services exist.

In this project an encoder for IM AF files will be designed, following the ISO 14496 Part 12: ISO base media file format and ISO 23000 Part 12: Interactive music application format. This file encoder will support a maximum of 16 simultaneously audio tracks with a sampling frequency of 44.1kHz at 16 bits per sample. In this version, individual music-tracks must be encoded in MP3. Also, it will be able to add different mixing presets and rules. Presets are predefined values of the volume of each instrument, presets will be very useful since a) the producer can create different versions of the track and b) users exchange and share their own mixtures. Rules are limitations imposed by the creator of the song, usually producers, to avoid users destroy the essence of the song. The encoder will be implemented in C and will have a simple command line user-interface in order to introduce the information needed by the program.

The technology presented in this project has to become the base for future applications. Having an encoder is the first step to create associated technology for the media sector. Interactive services will dominate the market in next years, so companies that will be able to adapt faster to the changes will have an important advantage in front of their competitors. Also, not only

companies will benefit from the encoder, researchers working in multi-track applications will make their job easier. They will move from working with multiple files (one for each track) to work with only one file.

The IM AF file encoder needs multi-track songs. Nowadays it is not straightforward to find versions of the songs in that format. In the future, as the standard becomes popular, studios will release their songs as IM AF files. In this project we will select examples of different music styles and features in order to demonstrate the good performance and compliance of the encoder. However, any song that satisfies the requirements is perfectly reproducible.

Besides the implementation in C of the encoder and its corresponding explanation, this thesis will compare previous interactive music services with the current model, demonstrating the effectiveness of the IM AF model. Also, it will study the file structure of MPEG-4 and MPEG-1 layer III formats, since the first one defines the bases for IM AF files and the second one is the supported audio file version (MP3).

Due to the short period for developing the thesis project, it will not be possible to implement the totality of the encoding standard. Hence, the project will include a section to describe reasonable improvements for future versions. However, the final version presented in this thesis will be functional on its own and the possible future improvements will be related to the incorporation of metadata as images or synchronized text (karaoke).

## **1.4 Contributions**

The concept of creating an interactive music service, where user can modify the volume of the instrument is not new. Since the year 2000, commercial applications have been originated in diverse countries around the world. For example, UK released the U-MYX, MXP4 and iKlax [3] in France and Music2.0 in Korea (these examples are explained in detail in chapter 2). These services supply various interactive tools for users based on multi-track music contents. However, each of them uses a different file format, letting no interoperability between them. Users cannot use the same song on those players causing fragmentation in the interactive music market.

IM AF file format has emerged to become the standard in the industry. It will provide interoperability between them. The fact that IM AF is based on ISO standards from MPEG-4 [4] makes it the perfect candidate to consolidate and consequently expand the interactive music service market. IM AF file keeps some functionalities from the previous systems, it works with multi-track songs and allows users to modify the volumes. Also enables users to save mixing presets and share them with other users.

Besides, IM AF introduces the concept of interactivity rules. Those rules should not be confused with Digital Rights Management. The IM AF rules are defined by the music/producer to avoid losing his/her artistic creation. For example, a composer might not want his/her guitar to be completely



eliminated or his/her bass jazz to be mixed as rock style. Rules can be of two types: selection and mixing. Selection rules applied to an instrument and specify which ones are active or inactive. Mixing rules allow driving the way the listener will interact with audio tracks volumes at rendering time.

Moreover, as an interactive application file, it has a specific space for metadata, information related with the album or the song is stored there. Also, images like the cover of the album or others can be stored there. It supports time text synchronized with the song, perfect for karaoke.

The IM AF file format supports a number of widespread media files formats. In terms of audio, it can handle uncompressed data (WAV and PCM) [5] or compress data (MP3 and AAC) [6] [7]. For images it works with JPEG [8] and for synchronized text uses the 3GPP standard [9]. Thus, it guarantees maxima compatibility. The significance of having a file that supports the most used file formats in the market ensures an easy transition for users to the new service. Any new costumer, from a producer that works with high quality audio files to a basic user that has his/her songs stored in MP3 files will find IM AF very useful.

The availability of an standard file like IM AF will encourage companies to focus on developing new tools without the interoperability fear. Recent history provides some examples of good technology that were not successful in the market and disappeared. Examples like DVD-Audio or MiniDisc failed in their purpose. The first one because it could not transmit to the public its quality benefits with respect to the CD, and the second because it wanted to have all the aspects under control (it worked with its own specific file format that is luck of interoperability).

With the description of the file and the corresponding encoder that will be developed in this thesis, it is intended to provide enough information and tools to help the popularization of IM AF files and its consequently the market expansion with interactive music services.

## 2. Background research and literature review

In this chapter is going to analyse the previous standards that have contributed to create the IM AF file format. Moreover, it will present similar applications based on interactivity files.

### 2.1. Overview of MPEG

In 1988 the International Organization for Standardization (ISO) together with the International Electrotechnical Commission (IEC) founded the Moving Picture Experts Group (MPEG) with the aim of creating a standard for digital formats of audio and video. The first format based on MPEG was MPEG-1 [10]; it is used for medium rates of data around 1.5 Mbit/s. This standard was followed by the MPEG-2 [11]; it is employed for high data transfer on the order of 10 Mbit/s or more. The last file format created by MPEG was MPEG-4 [4]; it is chosen for very low data transfer with rates of 64 Kbit/s or less.

All models have the same goal: to obtain a better quality through the use of complex encoding/compression methods. The combination of an array of pictures and sound tracks can represent a huge volume of memory occupied when displayed in digital formats. For example, a picture with a resolution of 360x288 pixels and 8-bit precision occupies 311 Kbytes. If the number of pictures per second is 24 and the uncompressed data is nearly 60 Mbit/s. In case of audio track, the data transfer is smaller; if the sound track is sampled at 44Khz with 16-bits, the data rate is 1.4 Mbit/s.

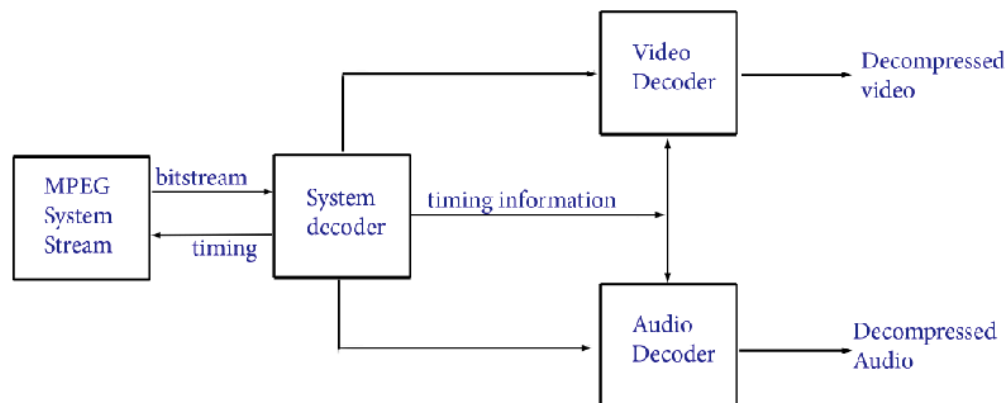


Figure 1. MPEG-1 system structure

The MPEG system splits the input data into layers in order to obtain a compressed version. Those layers have the assignment of mixing one or more audio and video bitstreams<sup>1</sup> into a single bitstream. A MPEG bitstream has two groups of layers: system layer and compression layer. The system

---

<sup>1</sup> A series of bits. It typically refers to the transmission of bits but may refer to bits in memory or in storage

layer offers a cover for the compression layers, and those store the audio and video data that will be later decoded. Also, the system layer furnishes the tools for de-multiplexing the interleaved compression layer. A block diagram used by most MPEG systems is shown in Figure 1.

The MPEG bitstream involves a string of consecutive packs that each one is break down in packets, as illustrated in Figure 2. One pack has a single start code and header followed by various packets of data. And one packet is conformed by a packet start code and header, succeed by packet data. This data represents compress audio or video samples.

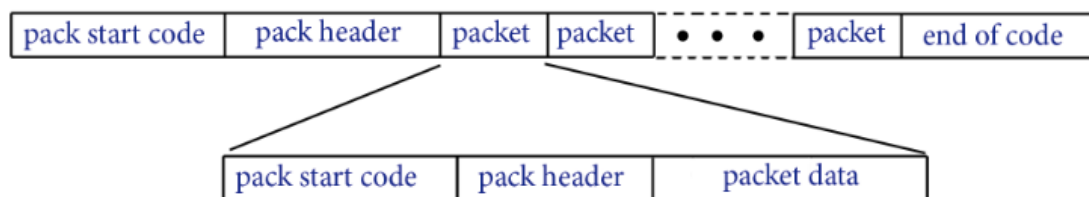


Figure 2. System layer pack and packet structure

The decoder analyse the bitstream and provides the partitioned video and audio data to the suitable decoders alongside timing information.

## 2.2. MPEG Audio Compression

Audio compression uses different tools to reduce and eliminate unnecessary data in order to have a smaller size, whereby the file will be more useful.

Perceptive coding is a technique used in audio that bases on the theorem of auditory masking. In presence of two nearby frequencies, the ear/brain combination is more sensitive to the strongest one. The weaker frequencies that are covered by the loudest one can be rejected. Consequently, the number of bits needed to codify the sound will decrease. The heightened quantizing error generated by reducing the number of bits is admissible if it is masked by the presence of the first tone. Another successful technique is the sub-band coding. It imitates the behaviour of the ear for analysing the different frequencies. For that, it divides the audio spectrum in multiples bands getting an exhaustive control over the frequency range. Then, those signals are quantized independently; reducing significantly the number of bits to transmit. A further method is Transform Coding, where the time-domain representation of the signal is transformed into a frequency domain such as Fourier, discrete cosine or wavelet transform [12]. Audio signals have slowly changes in amplitude along time and so the coefficients of the transform can be transmitted infrequently. Not all parts of the signal can be transmitted in that way; in presence of transients<sup>2</sup> adaptive systems are needed in presence of

<sup>2</sup> In audio, high amplitude and short duration sound.

transients. Transients contain important information of the sound, so their coefficients have to be refreshed constantly. While on the contrary, parts of the signal with stationary parts, such as maintained notes, the refreshing rate can be decreased. An example of this process is used in Layer-III of MPEG audio where the discrete cosine transform is used to code the samples.

### **2.3. MPEG-1 Layer-III**

MPEG-1 Layer-III, most commonly known as MP3 [6], was released in 1991 soon become the most used tool for Internet and audio delivery. It became very popular due to its big impact on the music industry, offering good sound quality with low bit-rate. The subsequent technology that has appeared since then has contributed even more to the popularity of MP3. Extensive uses of computers with fast Internet access and personal music players where users can store thousands of tracks have consolidated MP3 as the standard file format for the general public.

MPEG has designed a file that offers flexibility in order to be applicable in different scenarios. It works with mono, stereo and joint stereo signals. Stereo tracks are not very useful in terms of saving storage. Comparing to the same MP3 file, stereo tracks would need almost twice the size to provide the same quality as a mono file has. Opposite to that, joint stereo achieves efficient results in terms of saving space, coding parts of the spectrum that human ear cannot perceive as mono. It creates two channels, one containing the information of both channels and the other the differences. Thus, it succeeds with a lower bit file. However, this method fails when it has to codify sounds with phase shifted or delay effects. Those can be eliminated from the original sound or can cause interferences damaging the final sound.

While keeping the versatility of MP3, MPEG works on different sampling frequencies. MPEG-1 fixes three values: 32 kHz, 44.1kHz and 48 kHz. MPEG-2 decreases this to half rates: 16 kHz, 22.05kHz and 24kHz. Also, MPEG audio supports variable compression ratio. For Layer-III the standard determine a variety of bit-rates from 8 Kbit/s to 320 Kbit/s. The most used are 192 Kbit/s and 320 Kbit/s as they provide transparent quality. Bit-rate can change from frame to frame (variable bit-rate) permitting higher bit-rates for more complex parts of the song while less space is needed for less complex parts.

The MP3 encoder is shown in Figure 3. It is composed by a filter bank that divides the signal into 32 sub-bands. The perceptual model fixes the quality of the encoder, generating the permitted noise values for each frequency section. In MP3 these sections are the same as the critical bands of human hearing [13]. Finally, the generated data is quantized and coded. Quantization uses a power-low<sup>3</sup> system, where larger values are coded with less accuracy

---

<sup>3</sup> Mathematical relationship between two quantities.

[14], and then those values are coded using a Huffman coding. Different Huffman tables can be selected.

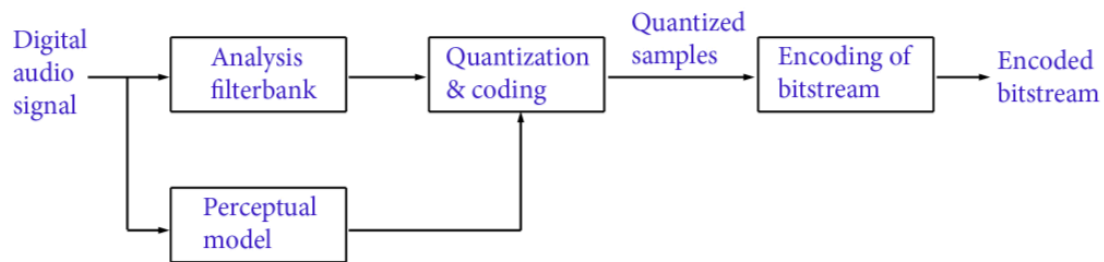


Figure 3. MPEG Layer-3 encoder

The fulfilment of the MPEG standard does not guarantee an optimal output in terms of sound quality. There are different factors that have an influence on the final result. MPEG Layer-III presents artefacts<sup>4</sup> caused by a time-varying error signal that appears at some frequencies. Another problem occurs when the block of music data to be encoded is larger than the actual bit-rate. In that case some frequencies, especially high frequencies, may be deleted.

Audio encoders may have different architectures depending on the model. MPEG standard encoders do not have any restriction in their design. However, all files produced by them have to be compatible with the corresponding decoder. MP3 file is split into small blocks or frames where the data is stored. Each frame has 1152 samples per frame [15]. The standard sampling frequency is 44.1kHz, which means that the duration of one frame is  $1152/44100 \approx 0.026$  sec. Knowing the bit-rate it is possible to calculate the size in Bytes of one frame [15], if the bit-rate is 128kbps the size is 417 Bytes, while if it is 192kbps the size is 626 Bytes. At beginning and end, MP3 files may have ID3 TAGs. Those are metadata containers allowing information related to the song as title, year of creation and style.

[ID3 TAG] Frame1 Frame2 Frame3 ... FrameN [ID3 TAG]

Each frame is structured in the same way. It has a 32-bit header followed by the samples. The header contains information associated with technical specifications as MPEG version, Layer, Bit-rate and sampling frequency. The decoder will use this information to decode the file properly.

As each encoder creates the file in its own way, the quality of the resulting MP3 file may vary. Different methods exist to analyse the performance of the encoders but well controlled listening tests with large amount of tracks and subjects is the best way to categorise their quality.

---

<sup>4</sup> A distortion in a sound caused by a limitation or malfunction in the hardware or software.

## 2.4 MPEG-4

MPEG-4 was created as an evolution of MPEG-2. Unlike its predecessor, improving the compression methods was not the main task. The growing availability of the bandwidth has permitted better transference of data. The main difference with previous audio-visual coding standards is the object-based representation model that consolidates MPEG-4. An object-based scene is created using independent objects that have connection in space and time. Different types of data are integrated in the audio-visual scene. They can be audio objects like multichannel audio content or speech, or video, like movies.

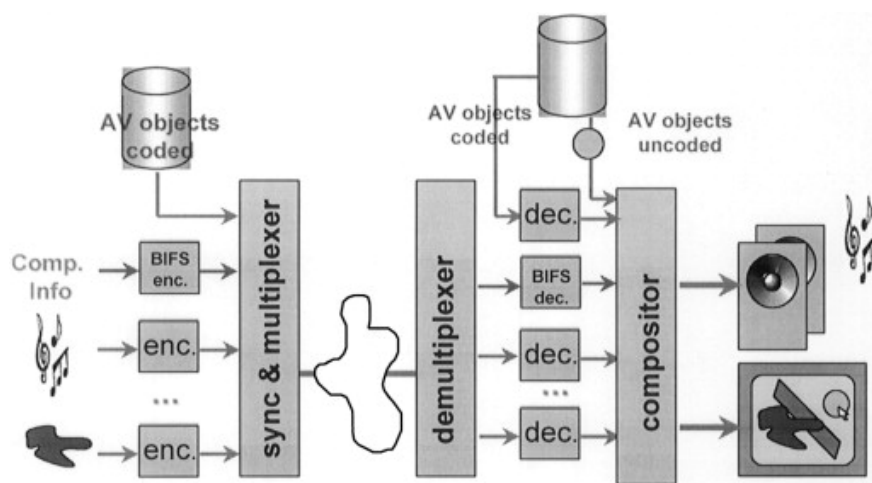


Figure 4. MPEG-4 object-based architecture [16]

An MPEG-4 system involves various audio-visual objects that are conveyed through a number of audio-visual streams to a receiver, those can be online or stored in a file. This section will provide a brief overview of the tools available in the MPEG-4 standard, focusing its interest on the Systems described in ISO14496 Part 1, and Audio described in ISO 14496 Part 3, as they are the most relevant related to the work described in the thesis. The first Systems tool is the Object descriptor framework [17]; it defines the relationship between the individual audio or video stream and the media objects in the scene. ODs supply information such as the elementary stream available to represent a given media object, the characteristics of the decoder needed to understand the stream and the location of the elementary stream data. This location can be a URL of the element allocated in the hard drive or in the web. Another tool is the Transport tool defines the MP4 [17] file format for storing MPEG-4 information (the MPEG-4 file format). The overall operation of a system communicating like audio-visual scenes is divided in two sections. The transmitter compresses the information of the scene

together with synchronization information. These data are multiplexed in one or more binary stream and finally sent or stored. The receiver reverses the process by demultiplexing and decompressing the information. The final user gets the media objects according to the scene information and may have the chance to interact with the presentation.

On the other hand, MPEG-4 Audio represents a new standard that combines multiple formats of audio: sounds created naturally with sounds generated by computer, low quality transmission with high bit-rate transmission, complex soundtracks with simpler ones, and traditional content with interactive content. Unlike previous standards MPEG-4 does not focus on a single application, it is used in every application that needs advance audio compression, synthesis or manipulation. Audio storage and transportation are not defined in the standard because there are a wide range of applications that can benefit of MPEG-4 technology, and it is impossible to describe a single solution that would cover all of them. What it is defined is an interface: the Delivery Multimedia Interface Format (DMIF) [17]. This interface allows transmission functions much more complex than its previous MPEG standards.

MPEG-4 Audio does not focus only on coding high quality audio at the necessary bit-rate as the previous standards did, it also includes new techniques for transmitting audio at low bit-rates, very useful for the present technologies as Internet or digital radio. Also, MPEG-4 provides different toolsets that can work independent or together depending on the application. Preceding standards transmitted individual content, MPEG-4 allows more than one content at a time, introducing the concept of soundtrack. Transmitting several audio objects with multiple tools creates an audio composition system or soundtrack. This ability furnishes MPEG-4 important superiorities in quality and flexibility versus its predecessors.

As mentioned above, the file format defined by MPEG-4 is called MP4, it is described in ISO 14496 Part 12 Base Media File Format. It is an important part of the IM AF file, thus it will be explained in detail in the next sections. It is intended to accommodate multi-media information in a versatile arrangement that allows ease manipulation, correction and display of the media. The structure of the file is based on boxes; a file can be decomposed into main boxes and each one has its sub-boxes and the structure of the boxes is deduced from their type. The file format is created with the intention of being independent of any pre-defined architecture while maintaining an efficient relationship between all the boxes.

## **2.5 Previous interactive music systems**

IM AF was not the first file format to include interactive control over the media. In this section we describe three examples of earlier formats: iKlax [3], IEEE 1599 [18], and iXMF [19].

**iKlax** technology was developed by iKlax Media company and LaBRI in France. iKlax proposes a file format with separated tracks and interactivity to manipulate the music piece. The project includes a music player and a music editor. iKlax format groups all the tracks of a song and related metadata in a single file, while tracks maintain quality as they do not suffer any compression (multi-track compression) because they are stored separately from the other tracks. iKlax file has two levels of interactivity, the first level allows the selection of the track, where listeners can chose the track that they want to listen. The second level is the mixing; here listeners can modify the level of each track. Each level has different constraints. Those are stored in the metadata section of the file. For example, the selection level defines two constraints: exclusion and implication. The exclusion constraint specifies that the selection of track *A* stop automatically track *B*; and the implication constraint specifies that the selection of track *A* activates track *B*. On the other hand, the mixing level defines three constraints: equality, inequality, and balance. The equality constraint means that some elements will have the same level during the whole song. Inequality means that some elements will be at a higher level than others. Finally, balance defines a group of elements with a constant level.

Together with the iKlax file, iKlax provides the Player (Figure 5). It is a classic media player that includes a zone to control the different tracks of the song.



Figure 5. iKlax Player

**IEEE 1599** combines in a single file music and XML symbols as graphical representation of the music or performance indications. This information is integrated and synchronized within the same framework and can be accessible individually or as a whole.

An exhaustive description of music must integrate different types of information. IEEE 1599 has developed a new XML encoding system to



allocate this heterogeneous information in a single file. XML organises the information in six different layers [18]:

- General – it stores information about the piece, i.e. title, author.
- Logic – coherent representation of score symbols.
- Structural – classifying musical objects and the connection.
- Notational – visual representation of the score.
- Performance – computer-based description of musical representation.
- Audio - digital audio recording.

The standard accentuates the readability of symbols by humans and computers, and it is created for applications that include additional information of the piece. Figure 6 shows the characteristic multi-layer structure of IEEE 1559. Additionally, the figure includes graphical examples to illustrate the objective of each layer.

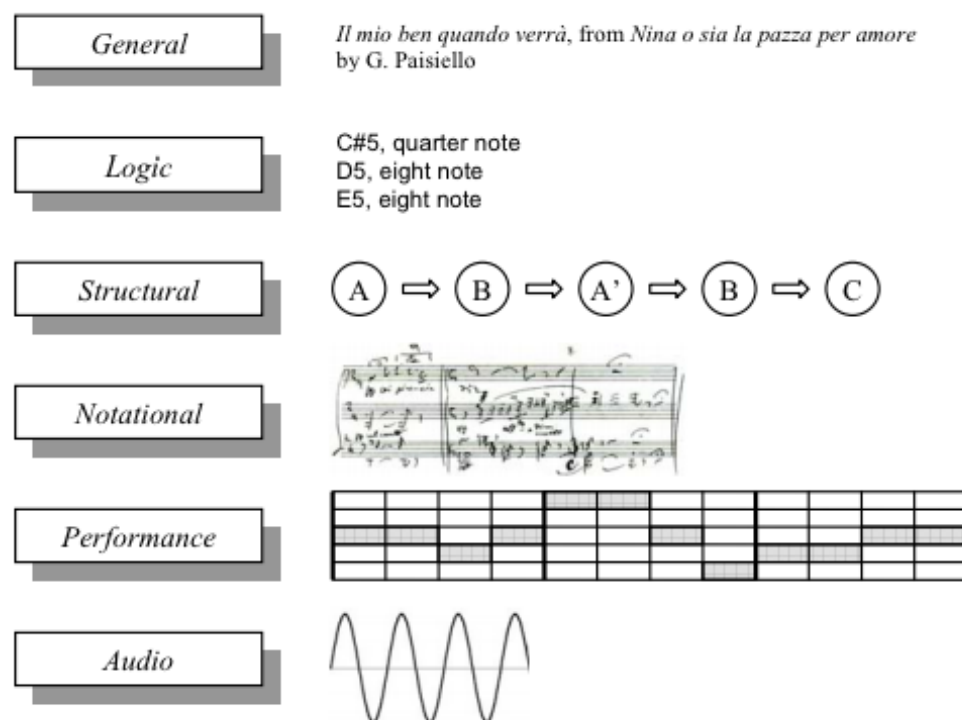


Figure 6. Multi-layer structured of IEEE 1559 [18]

**iXMF** or interactive eXtensible Music Format is a file format created by the video games industry, to provide a standard structured audio file format that supports cross-platform interchange of advance interactive audio tracks.

An iXMF file includes in a single file all the information needed to perform the audio track as the artist intended. The same information defines the structure of the file. iXMF uses a structure such that an event can be triggered at a particular moment in time . The selected event can activate a wide range of activities such as the playing of an audio file or the execution of specific code.

The structure of a single file is as follows [19]:

- Cue folder – contains all of the cue description resources. Each cue is a collection of Media Chunks.
- Media Chunks folder – contains all media chunk resources that may contain the same metadata files as cue.
- Media Files folder – contains all playable media files for the soundtrack
- Transitions folder – contains transition definitions resources as cross-fades and edits.
- PositionRules folder – includes position definitions as start at chunk beginning and start at next bar.
- Callbacks folder – includes callback definitions as cue end and chunk end.

The figure below shows an example of how a Cue Sheet might look.

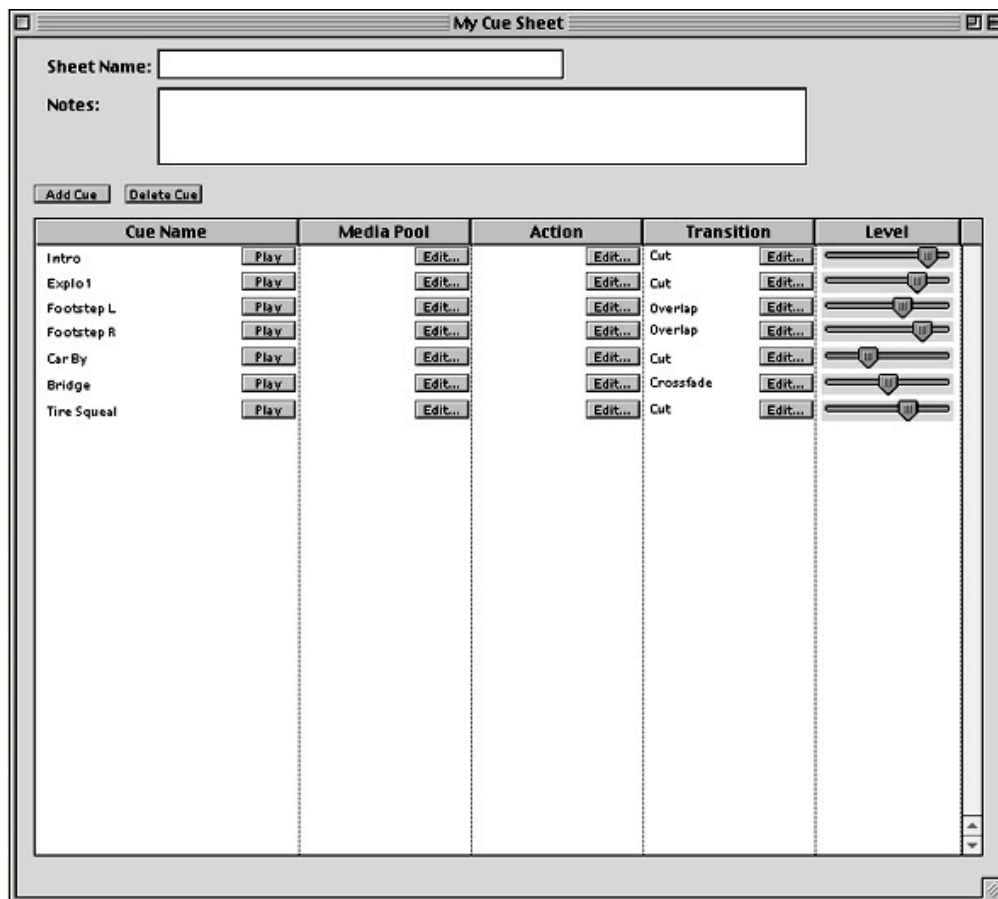


Figure 7. iXMF player [20]

### **3. Development of the IM AF encoder**

In this chapter is going to present the interactive music file IM AF. Also, it will analyse the different sections of the file that are used in the encoder developed in this thesis. IM AF uses parts of the MPEG-4 standard as new parts designed specifically for this file.

#### **3.1 Interactive music service**

Standard version of creating music content is to record the instruments in individual tracks and then mixing all the tracks generating the final version in stereo. The producer together with musicians decides the style of the song and the presence of each instrument in the song. The final user receives the song where he/she can only change the global volume, having no chances to adjust individual volumes.

Seeing this limitation, some companies started to develop new interactive services that allow users more control over the instruments (Chapter 2.5). To ensure interoperability between different multimedia content, the International Standard Organization (ISO) presented in 2010 the Interactive Music Application Format (IM AF). This standard is defined under the umbrella of Multimedia Application Formats (MPEG-A) Part 12. IM AF specifies how to connect multiple tracks with related information under a well-defined structure that facilitates the manipulation of interactive music content.

Interactive music content involves audio tracks, preset data and rules. Audio tracks represent the instruments of the song, they can be a single instrument or a group, sometimes is preferable to have the drums in a single track. Preset data is a pre-defined information related to the volume of each track, and allows the user to create different version of the song. Those values cannot be changed after the file is created, so it can be useful for producers to present the same track from different points of view. Finally, IM AF files introduce rules to avoid the user destroy the initial intention of the author. Moreover, additional media data can be used to improve the user's experience: timed text synchronized with the song for karaoke and images as cover of the album.

The created files are reproduced by an interactive music player as shown in Figure 7, users have two different options to listen the song: preset-mix mode and user-mix mode. In preset mode users chose one of the pre-defined presets in the IM AF file, and then the tracks change the volume according to the values of the preset. In user mode, the user selects/deselects the tracks and controls the volume of the tracks. All the actions performed by the user need to be compatible with the rules; otherwise the actions will not be carried out.

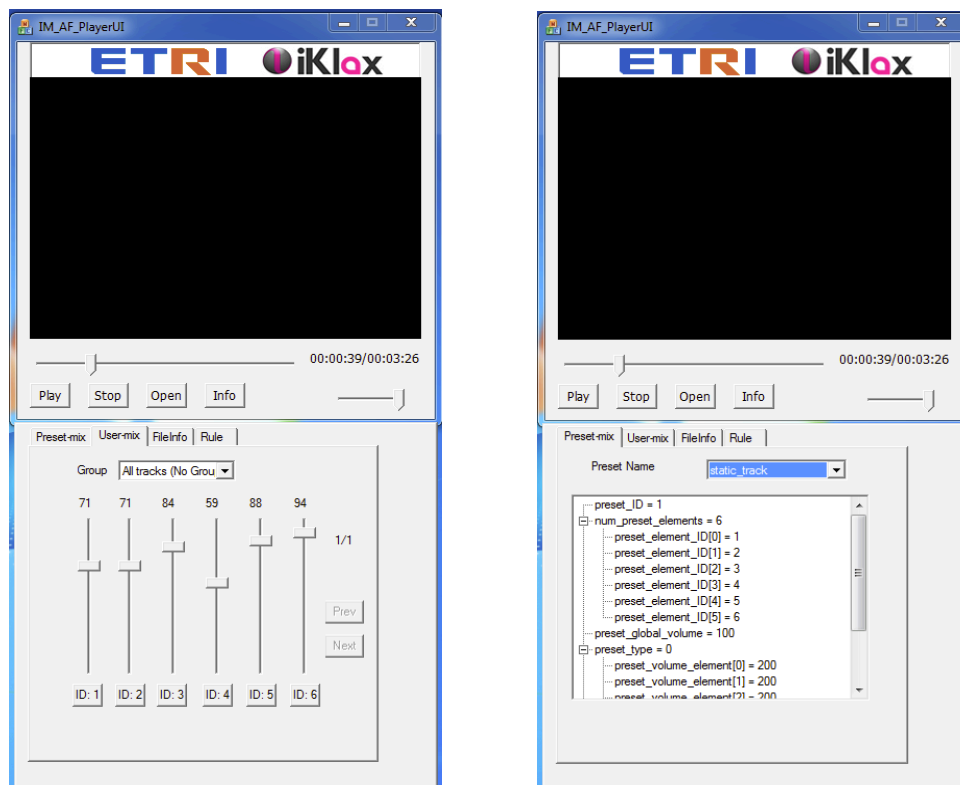


Figure 7 IM AF player. User-mix mode (left) and preset mode (right).

### 3.2. Creating an IM AF file

The encoder is responsible for creating the IM AF file, for doing that it follows the standard defined in ISO 23000-12. The framework of this file is based on the MPEG-4 ISO based Media File Format standard; IM AF has introduced some improvements to enable interactive control. In this section we will describe in detail the structure of the IM AF files together with the basic framework of the MPEG-4 used in IM AF.

IM AF files consists of a series of boxes that include all data. There are two different types of boxes: ones that may contain other boxes inside them, and others that just contain data (called FullBoxes). All boxes start with the header, which defines the *size* and *type*. FullBoxes incorporate in the header the *version* and *flag* information. The *size* defines the total *size* of the box, including data and, if necessary, other boxes. It has a *size* of 32 bits (unsigned integer), but if the data stored in the box is bigger than that it can use 64 bits. *Type* is the identification of the box, and each box has its own type, i.e. *ftyp*, *moov*, *mdia*. The *version* is an integer (8 bits) and specifies the *version* of the box, and the flag is a 24-bit integer and its use depends on each box.

A complete list of the boxes that form the structure is presented below:

ftyp					File type and compatibility
moov					Container of all the metadata

	mvhd					Movie header
	trak					Container for an individual track
		tkhd				Track header
		mdia				Media information container
			mdhd			Media header
			hdlr			Handler, declares media type “soun” (sound) for audio data
			minf			Media information container
				smhd		Sound media header
				dinf		Data information box
					dref	Data reference box
				stbl		Sample table box
					stsd	Sample description box
					stts	Time to sample
					stsc	Sample to chunk
					stsz	Sample size
					stco	Chunk offset
	grco					Container for groups
		grup				Group box
	prco					Container for the preset
		prst				Preset box
	ruco					Container for rules
		rusc				Selection rules
		rumx				Mixing rule box
mdat						Media data container
meta						Metadata

Table 1. Structure of IM AF file

This list represents the structure of the implemented file, ISO 23000-12 defines some more but they are not strictly necessary. Only one *movie box* shall be in the file, and it is placed nearby the top or bottom of the file to allow its easy location. The *file type box* ‘ftyp’ shall take place at the beginning of the file; thus it defines the file type. The rest of the boxes are allocated as proposed in [21]. Each box has a predefined structure written in the syntax description language (SDL)[22]. This language allows the easy conversion using C, C++ or Java . In this thesis the encoder has been programmed in C . A presentation is formed by various files. One file stores the metadata and the media data for the whole presentation. The other files are not needed to be part of [21] as they contain other media data or other information. These files are images, text or other formats and they have their own standard. In the case of IM AF files, the supported files are JPEG 2000 described in [8] and 3GPP synchronized text described in [9]. As our goal in this thesis was not the full implementation of the IM AF file format encoder but only a subset of it as

proof of concept, the latter features have not been integrated in this version. Nevertheless, their incorporation would be very interesting for future improvements.

The metadata is stored within the metadata folder 'meta'; the media data is contained in the same file ('mdia') or in other files. IM AF can handle tracks allocated outside the file, i.e. online or different folders. It uses the address or URI<sup>5</sup> to find the information needed to reproduce it. Each track has its own label (track identifier); it identifies the track during the whole process. When URIs are used as a track identifier, the URI must define the format and meaning of the data. If the URI contains a domain name (it is a URL<sup>5</sup>), then it should also contain a month-date in the form of mm/yy. This information must match the time of the definition of the extension, and the URI must be authorized by the owner of the domain according to the same date and month. This avoids problems in case the domain changes owner.

Finally, IM AF allows files to be reproduced in devices that are not able to decode several audio tracks simultaneously (backwards compatibility with legacy devices). The *flag* inside track header decides whether the player is able or not to reproduce multi-track files. If the *flag* is set to '1', then the track is enabled for multi-track operation. If the *flag* is set to '0' the track is disabled. Hence, the IM AF player decodes only tracks with *flag* set to '1'.

### 3.3. File structure of IM AF

This section will analyse the different boxes used in the IM AF file. It will follow the same order as in Table 1. Section 4 describes how the boxes are implemented.

First box is **File Type Box** ('ftyp'). It specifies the brand identifier of the file. Table 2 shows the different brands supported by IM AF files, where each one has different characteristics. The media files may be compatible with more than one brand. Therefore, it reserves space for compatible brands. Also, the file type box has a *minor version* variable for informative use only. This variable may allow more precise description of the *main brand*.

Brands	Audio	Max. # of simultaneously decoded audio tracks	Max. sampling frequency/ bits	Application
'im01'	MP3	4	44.1kHz/ 16bits	Mobile
'im02'		6		
'im03'		8		
'im04'		2		

---

<sup>5</sup> Uniform resource identifier/locator

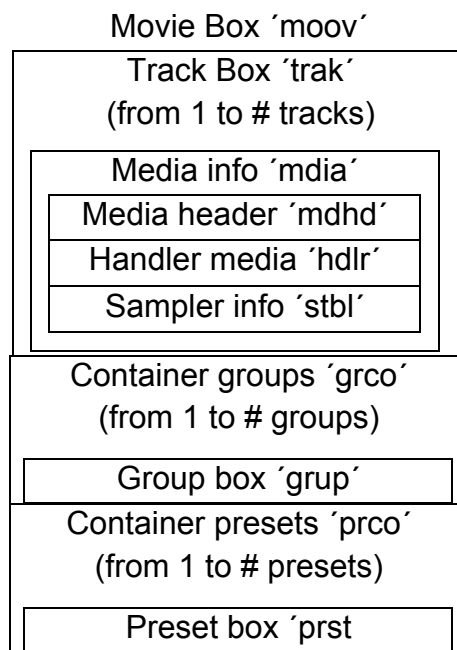
'im11'		16	48 kHz/ 16bits	Normal
'im21'		32	96 kHz/ 24bits	High-end

Table 2. Brands for IM AF

The 'im01', 'im02', 'im03' and 'im04' are intended for mobile applications because they have a lower resolution (less space in memory), and they differ in the maximum number of tracks that they allow. The 'im11' is intended for normal applications because it uses a bigger sampling frequency obtaining better quality. Finally, the 'im21' is intended for High-end applications as it uses a higher sampling frequency and resolution, the size of this files will be much more bigger (in bytes) than the mobile applications getting the maximum quality from the audio files.

Audio tracks are defined by samples of data. The **Media Data Box** contains all those samples. In the case of IM AF files, there will be one or more tracks stored in it. The structure of this box is very simple, it has an integer value that indicates the *size* of the box and a *type* values set to 'mdat'. Then, all the samples are store byte by byte. Afterwards, the sampler table box inside the movie box will contain all the information needed to decode the tracks.

Next box is **Movie Box** ('moov'), it is stored all the information that define the media data. This box does not contain any variables, just the *size* and *type* of the box. There is only one Movie Box in the file, and may include other boxes. The layout of the movie box is as follow:



Container rules 'ruco' (from 1 to # rules)
Selection rule 'rusc'
Mixing rule 'rumx'

Table 3. Movie Box structure

**Movie Header Box** is a Full Box and it specifies the characteristics of the entire Movie (only one movie header box in the file). The layout is:

Movie header box	Bytes
Box size	4
Type = 'mvhd'	4
Version	1
Flags	3
Creation time	4
Modification time	4
Time scale	4
Duration	4
Rate	4
Volume	2
Reserved	10
Matrix	36
Pre-defined	24
Next track ID	4

Table 4. Movie header box

*Box size* is an integer that specifies the size of the entire box; in this case the size is 108 bytes. The *version* is 0 and *type* identifies the box and must be set to 'mvhd'. *Creation* and *modification time* are two integers that specify the date in seconds. These values refer to the number of seconds since January 1<sup>st</sup>, 1904 as defined in the Coordinated Universal Time (UTC) [23]. The *time scale* refers to the whole presentation and it is expressed in time units per second. *Duration* declares the length of the longest track, and the units are according to the time scale. *Rate* and *matrix* are used when encoding video and *volume* indicates the main volume of the presentation. Finally, *next track ID* refers to the value of the track ID that will be added to the presentation. This value will always be the number of tracks plus one.

After describing the general characteristics of the media, it is time to specify the particular information of each track. This information is stored in the **Track Box**; there will be as many of these boxes as tracks in the file. Each track is



independent of the others and transports its own temporal and spatial information. Tracks contain media data, and there shall be at least one track in the file. Similarly as the movie box, the Track box does not contain any variables rather than the *size* and *type*. It is used to accomodate the different boxes needed to describe each track.

The first box inside the Track Box is **Track Header Box**. This box specifies the characteristics of a single track. Exactly one Track Header box is held in a track. The structure of the box is as follows:

Track header box	Bytes
Track size	4
Type = 'tkhd'	4
Version	1
Flags	3
Creation time	4
Modification time	4
Track ID	4
Reserved	4
Duration	4
Reserved	8
Layer	2
Alternate Group	2
Volume	2
Reserved	2
Matrix	36
Track width	4
Track height	4

Table 5. Track Header Box

*Track size* specifies the number of bytes in the box, in this case is 92 Bytes; the *type* must set to 'tkhd', and *version* to 0. *Flag* indicates if the track is enable or disable. A disable track is interpreted as if it was not in the file. *Creation* and *modification* time follows the same standard [23] as their similar movie header, but in this case they refer to a particular track. *Track ID* is an integer that identifies this track over the rest and its value must be unique. *Duration* represents the sum of all the samples and the units are expressed according to the time scale of the presentation. *Volume* specifies the relative volume of the track and is fixed to 1. The rest of the values are not used in this presentation because they refer to files with video tracks.

The media declaration container, or **Media Box**, describes and defines all the information about the media data in a track. Media Box is a container of other boxes such as the media type (video or audio); the media handler used to interpret the sample data and the media information that includes all the information related to times and samples.

The **Media Header Box** specifies the characteristics of the media in track, as *time scale* and *duration*. The structure of the media header is:

Media header box	Bytes
Media size	4
Type = 'mdhd'	4
Version	1
Flags	3
Creation time	4
Modification time	4
Time Scale	4
Duration	4
Language	2
Pre-defined	2

Table 6. Media Header Box

The *media size* specifies the size of the box; in this case the size is 32 bytes. *Type* must be set to 'mdhd' and *version* and *flag* are set to 0. *Creation* and *modification* time declares the most recent time that the media in this track was created following [23]. *Time scale* refers to this media and it is expressed in time units per second. In this encoder, both time scales (movie and media header) have the same value. *Duration* specifies the time of the media; and *language* declares the code of the language for this media. It is a three characters code defined in [24].

The **Handler References Box** declares the nature of the media track. For example an audio track would be handled by an audio handler. It is a Full Box so it includes the *size*, *type*, *version* and *flag*. The handler defines an integer called *handler type* that contains the value 'sound', as the media tracks used in this presentation are audio. Moreover, it includes a variable string called *Name*, which gives a human readable name for the track type. It should be noted that *name* has a variable size and is terminated by a null character.

**Media Information Box** stores other boxes that contain information related to the media track: sound, data and sampler box. The **Sound Media Header Box** includes audio information independent of the coding, and the same header is used for all tracks in the presentation. This box is a Full Box and has the standard variables: *size*, *type*, *version* and *flag*. The size is 16 bytes and type is set to 'smhd'. *Version* and *flag* are set to 0. Also, it includes a

*balance* variable that places the mono tracks into stereo; the normal value is 0 (centre), but it supports left (-1.0) and right (1.0) panorama of the track.

The **Data Information Box** includes objects that help the localization of the media information of the track. The tracks can be stored in the same file or in other parts (i.e. online). This encoder supports tracks that are in the computer, so the data information box will only have **Data References Box**. This box contains the counter of the actual entries together with the size, type, version and flag.

The IM AF encoder stores the media data in samples. This information is located in the media data container. In order to reproduce these data, the encoder needs a series of tables relating time, offset and data of the media samples. Those tables are held in boxes inside the **Sample to Table Box**. The first box is the **Sample Description Box**, and gives detailed information of the coding type used, bit-rate and decoder specific information. Table 7 describes the different boxes that are stored inside the **Sample Description Box**.

...				
stds				
	mp4a			
		esds		
			ES_Descriptor	
				Decoder Config Descriptor
				SLConfig Descriptor
stts				

Table 7. Sample Description Box

The **Audio Sample Entry Box** stores the technical specifications of an audio track. The *type* is set to 'mp4a' because the audio files stored in the IM AF file are converted to MPEG-4 Audio. This conversion is explained in Chapter 4 of this thesis. Then, it has a *channel count* integer that specifies if the channel is mono (equal to 1) or stereo (equal to 2), a *sample size* variable in bits that takes the default value of 16 and *sample rate*. In our work *sample rate* is set to 44100.

This box ('mp4a') contains an **Elementary Stream Descriptor Box** [25]. It is a required extension to the audio sample description for MPEG-4 Audio, and appears only when the codec type is 'mp4a'. The **ES\_Descriptor** transports all information related to a particular stream. In this case Audio stream has three parts as described next.

The first part is an integer called *ES\_ID* that defines the *elementary stream ID*. The second part is a group of optional extension descriptors that support future extensions. Finally, the third part consists of two structures that convey the specific parameters of the elementary stream: **Decoder Configuration**

**Descriptor and Sync Layer Descriptor.** The decoder configuration descriptor supplies information about the decoder type and resources needed for the associated elementary stream. Table 8 presents the structure of the Decoder configuration descriptor.

Decoder Config. Desc.	Bits
Tag	8
Length	8
Object profile indication	8
Stream type	6
Up stream	1
Reserved	1
Buffer Size dB	24
Max. Bitrate	32
Average Bitrate	32

Table 8. Decoder configuration descriptor

*Tag* is a byte (8 bits) and it refers to the type of this box. As it is an elementary stream, some values differ from the standard box structure. *Length* is the size of the box (size in bytes). *Object profile* indication is an indication of the object profile that needs to be supported by the decoder. In [25] there are all the possible values. In this case, it gets the value 0x6B reserved for new ISO standards as IM AF. The *stream type* presents the type of this stream (Audio stream = 0x05) [25]. *Upstream* is zero and the *Buffer size dB* indicates the size (in bytes) of the decoder's buffer and has the value of 14000. Finally, the decoder descriptor specifies the *average bit-rate* and the *maximum bit-rate*. Both take the value of 128 Kbps.

The SLConfig Descriptor defines the configuration of the synchronization layer for this stream and it is configured according to its needs. This encoder does not use any extra packet so it is almost empty. The *tag* is 6 and the *predefined value* is 2.

Next box in the IM AF file is the **Sample Table Box** ('stts'). This box generates a table that links the decoding time with the sample number. It stores for each entry two values: *sample count* and *sample delta*. The first one refers to the number of frames that have the same duration and the second value is the duration of those frames (in milliseconds). The frames of MP3, as indicated in Section 2.3, have duration of 26ms. It also includes an integer that defines the number of entries to the table ('entry\_count').

Entry count	1934
(count, delta)[0]	1,0

(count, delta)[1]	7,26
(count, delta)[2]	1,27
...	...
(count, delta)[1933]	7,26

Table 9. Sample Table

Table 9 shows the structure of the sample table. Although theoretically the duration of each frame is 26ms, in practice the duration is 26.125ms. This causes that it has to add a frame of 27ms each 8 frames. Because 8 times 26.125 is 209, similarly as 7 times 26 plus 27 is 209.

Samples within the media data are stored in frames or chunks; chunks may have a variable size. The **Sample To Chunk Box** ('stsc') creates a table that includes the size of each chunk and the number of chunks that have the same size. Audio files used in the encoder have a constant bit rate. Thus, all the frames are equally sized and the table includes one entry. This table has three variables: the first one specifies the index of the first chunk (the first chunk of the track has the value 1); second is the number of samples sorted in each chunk and third is the index of samples entries. **Sample Size Box** ('stsz') holds the number of frames in the track and a table presenting the size in bytes of each frame (418 bytes). Table 10 shows the structure of the table. Although the frames should have the same size, in practice it can accommodate frames of sizes 417 and 418.

Sample count	7890
sample_size[0]	1,0
sample_size[1]	7,26
...	1,27
sample_size[32]	417
...	...
sample_size[7889]	418

Table 10. Samples size

The values presented in Table 9 and 10 are specific for an audio file (MP3) with a constant bit-rate of 128 Kbps. These values will change if another file is used. Finally, in order to identify the different tracks stored in the media data container, the **Chunk Offset Box** ('stco') indicates the position of the beginning data of each track.

Until this point, the boxes described here are common for MPEG-4. From now on, the boxes are specific of IM AF files. Both are essential for the smooth running of the file. The first box is the **Group Container Box**. It includes an integer that specifies the number of groups hold in the file, and as many

**Group Box** as the integer required. An individual group box has: a *group ID* that identifies the group, the *number of elements* involved in the group, and the *group activation mode* that describes the activation mode of each element inside the group. The second box is the **Preset Container Box** that allows users to define one or more presets. A preset may have all the instruments except the vocals for karaoke, or just the vocal and chorus for an acappella preset. Each preset is stored in a **Preset Box**, and has the following structure:

Preset box	Bits
Preset size	32
Type = 'prst'	32
Version	24
Flags	8
Preset ID	8
Num. preset elem.	8
Element ID	32
Preset Type	8
Global volume	8
Element volume	8
Name	14

Table 11. Preset box

*Flag* is an integer that defines how the preset will be displayed in the screen and if the user will be able to edit it [1]. The *preset ID* assigns a unique value to each preset; the number of elements in the preset is stored in *Number preset element* and the *element ID* contains all the identification value of each element of a concrete preset. There are two types of preset: static track volume and dynamic track volume. In the first one, the volume remains constant throughout the track; while the second, the volume of each element changes according to a certain period of time (time variant). The *global volume* indicates the volume of the whole preset, and the *element volume* designates the volume of each audio track. Finally, the preset box defines a human readable *name* to identify the preset.

When an IM AF file is created, users have interactive control over the song. The different choices that users make are filtered by the **Rule Container Box**. Two rules are defined: **Selection Rule Box** and **Mixing Rule Box**. Both have the same box structure; the first group of rules involves the selection of tracks and the second category is associated to the audio mixing.

Rule box	Bytes
Size	4
Type = 'rusc'/'rumx'	4

Version	3
Flags	1
Rule ID	2
Rule Type	1
Element ID	4
Key Element	4
Rule Description	2

Table 12. Selection rule box

*Rule ID* identifies the kind of rule and the *rule type* specifies the rule type. There are four types of selection rules: *min/max rule*, *exclusion rule*, *not mutes rule* and *implication rule*. The *min/max rule* specifies the minimum and maximum number of elements (tracks or groups) that can be reproduced (active state). The *exclusion rule* designates various elements that will never be in the active state at the same time. The *not mute rule* determines an element always in the active state. The *implication rule* defines that the activation of an element implies the activation of another element. On the other hand, the *mixing rule* defines four types of rules: *limit rule*, *equivalence rule* and *upper and lower rule*. The *limit rule* fixes the maximum limit of the volume of the track. The *equivalence rule* defines an equivalence relationship [1] between two volumes. The *upper/lower rule* applies to the volume of two tracks. An element A will have always an upper/lower volume than B.

*Element ID* represents the ID of the element affected by the rule. The *key element* identifies the element on which the rule is imposed. The last parameter is the *rule description*. It provides a human readable description of the rule. Last box in the file is the **Meta Box**; it stores descriptive or annotative information. This box contains a *handler box* that defines the format of the 'meta' box. The metadata can be in this file or located outside (URL).

## 4. Implementation

After describing the different parts within the IM AF file, this section will describe the code responsible of creating the encoder. Firstly, we will explain how to use the encoder and how it works. Secondly, we will analyse in detail the main programmed functions of the encoder. Finally, some tools that have been useful in order to facilitate the creation of the file will be presented.

### 4.1. Design: How it works

The encoder that creates the IM AF file has been implemented in C. It does not use any external library, ensuring that it will work easily in any computer. The program is composed of two parts: the main program and the IM AF header. The main program includes the functions to create the file and the header defines the structure of the boxes. Both parts will be analysed in Section 4.2. Moreover, there is a basic command line tool to let users introduce values such as the number of tracks and their names. The program has a static structure; meaning that the maximum number of tracks, presets and rules are defined by a global variable. The value is 10 for each one. In case the user would need more, it would be as easy as to change the value to a higher one.

Right at the beginning, the encoder asks the user to introduce the number of tracks that the IM AF file will incorporate. Then, the user writes the names of all the tracks that there will be in the file. The encoder detects if the track exists or not. If it does not, the program will ask again for the name. Those names are saved in memory because the encoder will need them afterwards. When the user has finished to introduce all the tracks, the encoder creates the binary file that will store all the information. The first function called in by the program is the *filtypebox*. This specifies the different types supported by the encoder. The major brand is *'im03'* and it allows the simultaneously decoding of 8 audio tracks (enough for this project). The compatible brand is set to *'isom'* because the file is an ISO base media file. Once the type is clear, the encoder transform the audio files (MP3) into MPEG-4 audio files. For doing that, it uses all the boxes described in section 3.3.

The first step is to extract the samples of the audio file; the function in charge of this is the media data function. It creates the media data box. This function reads each track, in this case an MP3 track, subtracts the samples and stores them in the binary file (Figure 8). It uses the names of the tracks introduced previously by the user, in order to open the files one by one. This process will be described in Section 4.2, but the idea is to find the beginning of the first frame and read the data until it reaches the end of the file. The intention is to avoid the ID3 header, as it does not contain useful information for our purpose.





reused by the *time to sample*, *sample to chunk* and *chunk offset* boxes. After loading all information on the *sample table box*, it is the turn of the rest of boxes in the *track box*.

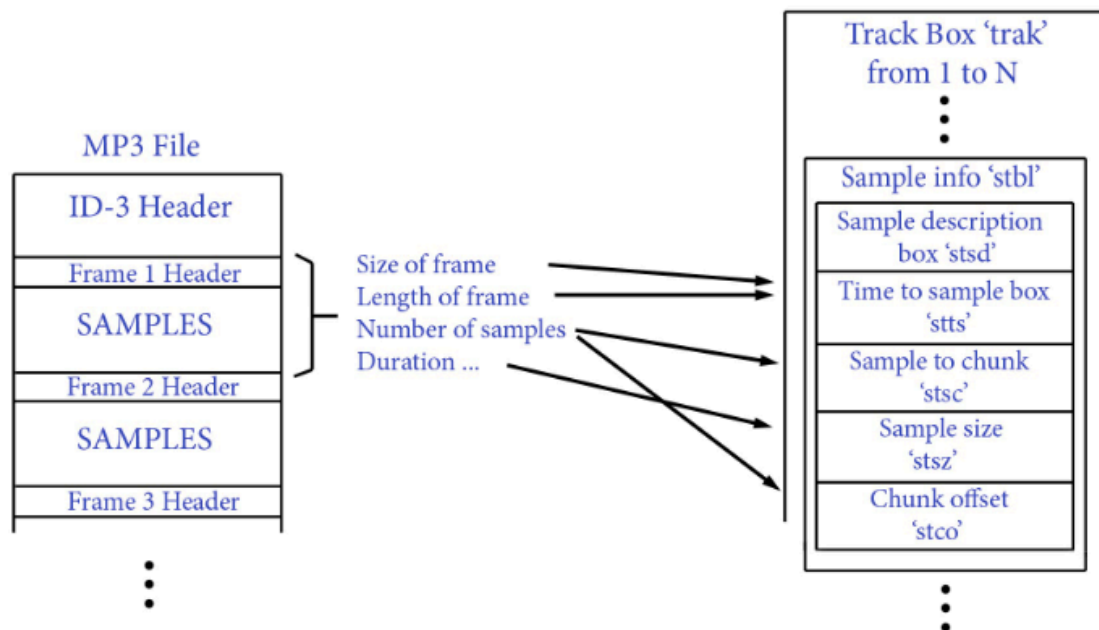


Figure 9. Sample data information

The *preset container* is next implemented after the track box. It creates a static preset with a fixed value. First, it defines the number of presets that there will be in the file. Then, it assigns the volume of each track; initially takes the value of 100. Changing this value will modify the volume of each track in the specific preset. The next step is to create the rules. The *rule container* creates two rules: one selection rule and one mixing rule. Finally, the *movie box* is created together with the *movie header box*. Here the encoder puts together the sizes of all boxes within the *movie box*. Doing that at the end of the code ensures that the total size of the *movie box* will match the sum of each one.

## 4.2. Implementation

This section will explain the most relevant parts of the code, so that a first time user will find answer to his/her doubts. The encoder is programmed in two files: *main.c* and *IM\_AF Encoder.h*. First we will analyse the header and then the main program. Before getting into the explanation of the files, it is necessary to address some issues that affect the entire program. It has been written in standard C, using a Mac OS X version 10.7.4 Intel Core 2 Duo with 2.1GHz. This processor writes the data in memory backwards. Therefore, it writes the wrong data in the file. To solve this problem, a small function has been implemented that swaps the order of the bytes. This function is called

every time a value has been saved in memory. In this manner, it ensures that the correct data is stored and written in the IM AF file in the correct order.

#### 4.2.1. Header: IM AF Encoder

The header includes the structure of the IM AF file. It has been implemented using the structures defined in [1] and [21]. Each box described in Section 3.2 represents a structure in the code. This type of storage allows us having a structure inside another one and so on; therefore yielding a very useful model. The general boxes as *file type*, *movie box* and *media data box* are defined as *typedef struct*, the rest are just structures. There are different types of data: integers that use 32 bytes, shorts that use 16 bytes and charts that use 8 bytes. Integers are the most used data, while charts are used to declare arrays of characters for declaring names or descriptions that the user may need to read.

There are some structures that will appear more than once in the IM AF file, like *Track Box* and *Preset Container*. The number of *Track Box* in the file will be less or equal than the maximum number of tracks specified by the *main brand* declared in the *File Type Box*. On the other hand, *Preset Container* will have a limit capacity specified by a global variable. It may hold a maximum of 10 items. This does not mean that the file does not support more; changing this constant, it will increase the limit of items.

#### 4.2.2. Main programme

The main file is segmented in various functions. Together they create the standard file. Below we list the main functions and their type.

Type	Function
int	main
void	filltypebox
int	mdatbox
int	trackstructure
int	samplecontainer
int	sampldescription
int	readTrack
int	presetcontainer
int	rulecontainer
void	Moovheaderbox
void	writemoovbox

Table 13. Main functions

The *main* function opens the file according to the name that the user has written through the command line. The encoder needs to localize the tracks in the computer. For doing that, a string containing the path directory of the folder that stores the songs is declared. If the songs are not in this folder, the encoder will not be able to find them. In case the user wants to use songs from another directory, he/she will need to change the address of the path directory in the *main* function and in the *readTrack* function. In these two places the encoder opens the audio tracks. The name entered by the user is stored in a structure called *nametrack* in order to use it again outside the *main* function. The other file opened by the *main* function is the IM AF file. It is a binary file and it is created in writing mode, while in the case of the audio file it opens the file in reading mode.

The audio tracks in this file are MP3 and have a determined structure. Once they are opened, the encoder needs to read the information. There are two functions where these files are analysed: *mdatbox* (it creates the media data box) and *readTrack* (it extracts the information for the sample tables). In the case of *mdatbox*, the reading is less accurate than in *readTrack*. The MP3 is characterised for having a frame structure and each frame has a header (described in Section 2.3) that is used as a reference for searching the frame. The *mdatbox* receives the file pointer of the song. The task in this function is to localize the beginning of the first frame of the MP3 and copy all the data into the IM AF file. It uses a loop with four conditionals ('if') for scanning the audio file with the intention of detecting the header. The header is formed by 32 bits that have a determined value [15]. The first 4 bits are always 0xFF and the other 24 bits vary between two or three values depending on the characteristics of the MP3. To ensure the proper functioning of the encoding, we have taken into account all options. The loop reads the MP3 in groups of four bytes each time. If the loop finds the header, it saves the position of it and jumps out of the loop. If it does not find it, it moves back three bytes and keeps reading the next four bytes. Thereby, it ensures that all the data of the MP3 is properly read. Once the loop finds the position of the header we need to introduce a correction, as the position is saved 4 bytes after the header. In order to get the correct position, the encoder must subtract 4 units from the variable that contains the position. Finally, the MP3 is read from this position until the end of the file and the information is stored in the IM AF file just after the size and type of the *media data box*.

Unlike the *mdatbox*, the scanning of the MP3 in *readTrack* is more elaborated. It uses the same method for searching the header of the frame, but it does not jump out the loop when the encoder finds the first header. It reads until the end of file and searches all headers. Each time it finds one, it stores in an array the length in bytes of the header. For this, it saves the position of one header and the position of the next one; with this information it calculates the length subtracting the second position from the first. This array is used to create the *sample size table*. The size of this array is 9000, it supports tracks

duration of 3.5 min: in case the user wants to use a longer track, he/she must increase the size of the array. In addition, it uses a counter to hold the number of samples in the whole MP3. For generating the *time to sample table*, the encoder creates a table of length equal to number of samples and stores in each row the length in milliseconds of each frame. As mentioned in Section 2.3 the duration varies between frames (26ms and 27ms).

Each time a function uses a structure from the header, the structure is passed as references. This means that the structure may be modified inside the function, and the changes made there will last once the function ends. The *writemoovbox* does not have the structures by references because its content is not modified. It just reads the information. This function writes in the correct order the structures into the IM AF file. It uses the function *fwrites* that specifies the size of the variable. It allows writing the data very precisely. The variables with size 32 or 16 bits are written straight into the file. However, variables having sizes of 14 or 17 bits are written bit by bit, so there are no free spaces between variables. Once the file is written with the correct information, the encoder closes the IM AF file and let it ready to play.

### 4.3. Programmes that analyse files

In the process of creating the IM AF file we have used various programs that analyse the structure of the file. Specifically programs that show the boxes of the MPEG-4 file, or in this case they work for IM AF too. The creation of a file like the one created here is full of little parameters that can lead to malfunction of the file. The order of the boxes is very important, so it is the size of each box. These programs have been an important part of the success of the encoder. The first program used was Mp4Explore [26]. It is MS Windows base software and shows the structure of MP4 files. Also it is free. Here there is a screen shot of this program for analysing an IM AF file.

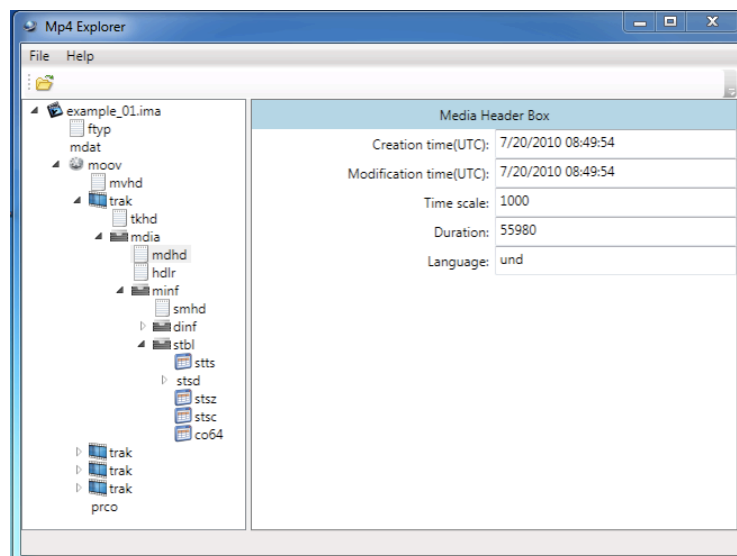


Figure 10. IM AF file in Mp4 Explorer

This software gives a very clear idea of how the IM AF file is structured. It details the content of all boxes, except the ones used only in IM AF, like *preset* and *rules* boxes. Because this software was created to work with ISO 14496 files (MPEG-4) so it does not support the new features of IM AF. Another defect that this program has is that it does not indicate that size of each box. Consequently, it is difficult to detect where the error is. For this reason the MP4 Browser [27] was used. This is another free software created for Windows with the same intention: to analyse MP4 files. But this one it does works with IM AF.

Below we show two screen shots of the MP4 Browser software.



Figure 11. IM AF file in MP4 Browser

Figure 11 presents the general structure of the IM AF file; more specifically, it shows the *file type box*. Unlike the previous software, the MP4 Browser presents more information: the type of the box and, more important, the size of it. Figure 12 shows an example of the *Time to sample Box*.

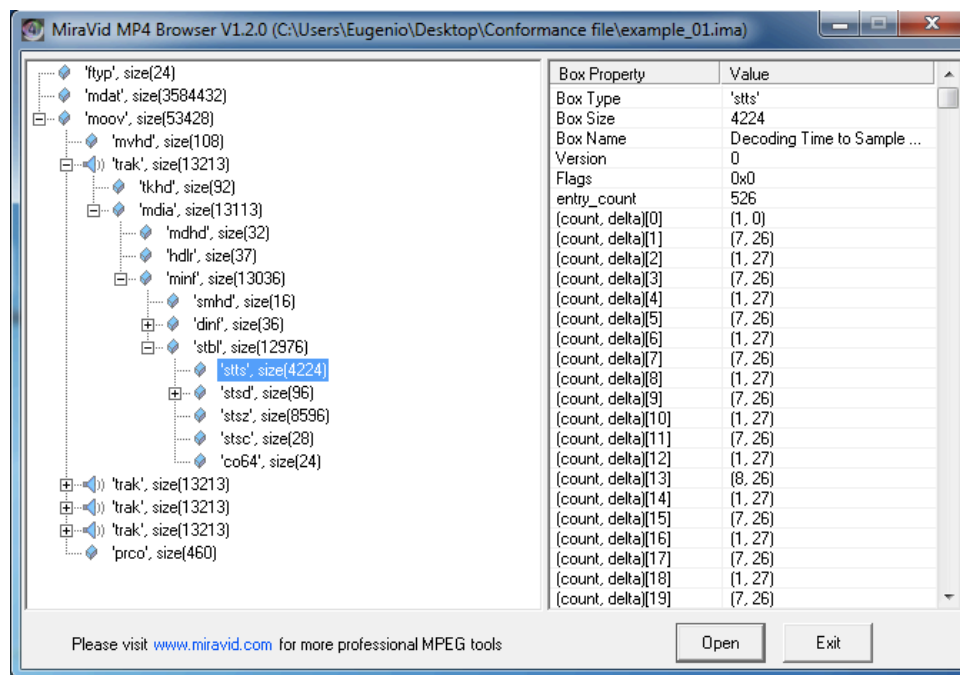


Figure 12. 'stts' box in MP4 Browser

In this case it presents all the possible information of the box: the number of entries and the value of each one. Like the MP4 Explorer, this software does not show the specific boxes implemented in the IM AF standard; but at least it indicates the size of them. This program makes the implementation of the file easier, because when a new box is created, the file is opened using this software and then, the engineer can check if the code is right or there is a problem. Without this type of programs the task of building an IM AF encoder would be much more difficult for the programmer. Both programs are very intuitive and simple to use. We note however, that they do not allow to modify or change any parameter of the file; they just open the file and present its structure.

In addition to these programs, it was used the *code.soundsoftware.co.uk* web page to create a project online called *IMAF Encoder*. This web is intended for researcher in the audio and music community, and allows you to upload the code into the repository (online folder) and access it from different computers. Using a Mercurial client [29] people can consult and modified the code stored in the repository without downloading it into his/her computer. Also, it is very easy to upload new material for keeping the project up to date. Thereby, it was worked during the programming stage in a very efficient way using best practice in software development as my tutor and other colleagues could follow my progression and add some comments.

Once the project will be finished, it will be available in the project page for everyone how will be interested, increasing the repercussion of the project and helping the popularization of the IM AF file.

## 5. Results and evaluation

In this chapter we will analyse the characteristics of the files created by the encoder. Three files have been generated despite that the encoder is very flexible regarding the way files are developed. In addition, we will describe the efficiency and performance of the software, as well as its weaknesses.

When the user wants to create a new file, he/she introduces the information through a command line interface. Before presenting the conformance files, we will explain the steps for doing that task. First, the encoder welcomes the user and asks for the number of tracks that there will be in the IM AF file.

```
Welcome to the IM_AF encoder
This program will allow you to create an IM_AF file.
How many tracks there will be in your IMAF file?
6
```

Figure 13. Creation of file – Step 1

In Figure 13 the user has selected 6 tracks. The maximum tracks that the IM AF file supports is 8 because the *major brand* in the *filtetypebox* is 'im03'. If this number is not enough, this value can be modified. The maximum number of tracks that the file can decode simultaneously ('im11') is 16. For avoiding the failure of the program, the encoder checks that the number of tracks introduced by the user is compatible with the value specified by the *major brand*. Figure 14 shows what happens when the user introduces a higher value.

```
Welcome to the IM_AF encoder
This program will allow you to create an IM_AF file.
How many tracks there will be in your IMAF file?
10
Sorry, for this version the number maximum ot tracks is 8
How many tracks there will be in your IMAF file:
3|
```

Figure 14. Checking the maximum number of tracks  
in the picture "the maximum tracks is 8"

After specifying the number of tracks, the next step for the user is to enter the name of each track. As mentioned before, the audio files must be in the same path as is declared in the code. The names of the tracks have to be written together with their extension. Figure 15 shows an example.



```

How many tracks there will be in your IMAF file?
3
Name of the track number: 1
bass.mp3
Name of the track number: 2
drums.mp3
Name of the track number: 3
synth.mp3

```

Figure 15. Creation file – Step 2

If the user has not written correctly the name of the file, or the encoder is not able to detect the track in the folder, it will ask again for the name. Figure 16 shows this situation.

```

How many tracks there will be in your IMAF file?
3
Name of the track number: 1
bass
Name does not exist. Try again:
bass.mp3

```

Figure 16. Checking the track's name

When the user has finished entering all the names, the encoder creates the preset defined previously in the code. To provide clarity on what is being written in the file, the encoder prints on the screen the characteristics of the preset. Figure 17 below shows how this information is presented.

```

Presets:
Static track volume preset: invariant volume related to each track
-----
Preset number 1: static_track
Enter volume for bass.mp3 = 120
Enter volume for drums.mp3 = 100
Enter volume for synth.mp3 = 80

```

Figure 17. Creation file – Step 3

The last step is to write the rules into the file. They are also written directly in the code rather than using the command line. The encoder shows the different rules implemented in order to make it clear to the user what is inside the file. Figure 18 shows how the information is presented.

```

Rules:
Rule 1: Not mute for channel 3
Rule 2: Upper rule between channel 1 and 2

File is created successfully, and ready to use!

```

Figure 18. Creation file – Step 4

We note that the rule is the last box. After that, the encoder writes all the information into the file. Thereby, the only thing that remains to be done is to inform to the user that the file is created. A file with extension IM AF ready to play in the IM AF player has been created.

## 5.1 Conformance files

The objective of this thesis has been to create an encoder able to generate IM AF files. To demonstrate its operation, three different files have been implemented. Each time a new IM AF file is presented we supply various examples to prove the efficiency of the encoder. For that purpose, we have selected three different multi-track songs from [28] with different styles. The criterion for choosing the tracks is to cover different music styles. Acoustic, electronic and rock are the three styles selected. Each song has different configurations of the encoder parameters. The figure below describes the characteristics of the IM AF files chosen.

Example 1	
Name:	Example_Acoustic.ima
Format:	MP3
#Instruments	6
Presets	Yes
Rules	Yes (2)

Type/Nº of instrument	Value of Static Preset	Selection Rule: Not mute	Mixing Rule: Equivalence
ID 1 - Drums	40	-	Element involved
ID 2 – Bass	80	-	Key element
ID 3 – Voice	120	Element involved	-
ID 4 – Guitar	180	-	-
ID 5 – Clarinet	200	-	-
ID 6 – Accordion	240	-	-

Table 14. Conformance file 1

The first example has six instruments, two rules and one static preset. The *selection* rule is *Not mute* and the element that will be always in active state is number 3: the voice. The rest of the instruments can be mute. The mixing rule is *Equivalence*, meaning that two elements will have the same volume during the whole piece of music. In this case the elements involved are the drums and the bass. At no time one element will sound higher than the other. The preset defines a static volume for the instruments; the secondary elements like the clarinet or accordion get more presence and the important ones like the drums or voice reduce the level.

Example 2	
Name:	Example_Electro.ima
Format:	MP3
#Instruments	6
Presets	Yes
Rules	Yes (1)

Type/Nº of instrument	Value of Static Preset	Mixing Rule: Upper
ID 1 - Drums	120	Element involved
ID 2 – Bass	100	Key element
ID 3 – Synth	80	-
ID 4 – Guitar	60	-
ID 5 – SFX	40	-
ID 6 – Loop	20	-

Table 15. Conformance file 2

The second example has six instruments, one rule and one static preset. The *Upper* rule defines that the bass will have an equally or higher volume than the drums. Thus, the bass will not be masked by the drums. The preset defines a static volume for the instruments. In this case the lower elements like the drums and the bass will have more presence in the song as in electronic music the low frequencies are more important than the higher ones.

Example 3	
Name:	Example_Rock.ima
Format:	MP3
#Instruments	5
Presets	Yes
Rules	Yes (2)

Type/Nº of instrument	Value of Static Preset	Selection Rule: Not mute	Mixing Rule: Lower
ID 1 - Drums	40	-	-
ID 2 – Bass	80	-	-
ID 3 – Voice	40	-	Element involved
ID 4 – AcGtr	100	Element involved	-
ID 5 – ElecGtr	104	-	Key element

Table 16. Conformance file 3

The third example consists of five instruments, two rules and one static preset. The lower rule defines that the voice will have an equal or lower volume than the electric guitar. In this type of music (indie/rock) the guitars have the same importance as the vocals, differently from Pop or Acoustic music where the voice has to be above the rest of the instruments. The other rule is not mute and the element involved is the acoustic guitar, to ensure the presence of this instrument during the whole track.

These three examples have been created to give an idea of what the program developed in this thesis is capable to do. More combinations can be made, but the three examples selected are a good representation of the possibilities of the new IM AF encoder.

## **5.2. Advantages**

In this thesis an IM AF encoder has been implemented that creates a new interactive file. It is intended for all users, from a beginner that does not know too much about programming to a professional engineer who wants to personalize the encoder to suit his/her needs. The beginner will find the encoder very useful and intuitive because he/she does not need to modify the code for generating an IM AF file. There is a command line interface that guides the newcomers through the creation of the file. This tool allows to enter the instruments in the file. The customization of the encoder by a user is a relatively simple task. The code is written in C, thus providing a classical programming environment very easy to understand. On the other hand, an experienced user working with the new encoder will find it sufficiently robust for his/her work; thus making it easy to make of the encoder a professional tool. Also, in case he/she would want to introduce new improvements in the file, the program created will represent the framework of his/her work and will be flexible enough to support the changes.

The encoder is very efficient. It is able to work with long audio files (5/6 minutes per track or more), open them, extract the information and merge it all inside the file in less than a minute. It does not need a powerful computer neither external libraries nor plugins. The two code files delivered with the thesis are the only tools needed to create the IM AF file. Each time a user introduces the name of the instrument, the encoder automatically writes the information in the corresponding box, thereby reducing considerably the amount of time that the user has to wait. Moreover, the lightness of the encoder, regarding memory consumption, makes it suitable for applications that work in low processor environments like smart phones or tablets.

The resulting file has a respectable size because it supports MP3 as the standard audio file. The size is quite problematic in a file that works with multi track files, as it readily increases the whole size. If the file works with four or five instruments with a non-compress format like 'wav' the final size will shoot up. Using MP3 reduces the size two or three orders of magnitude.

### 5.3. Weakness

This encoder has been created in a short period of time. Thus, there are some aspects that could have been done differently. The command line interface is limited; the user can introduce information related to the number and name of the tracks. The presets and rules information must be written directly in the code, hence making it troublesome for someone who does not know the C language. It would have been more comfortable for the user if the interface was an application by itself, rather than being part of the C compiler. A user who wants to create an IM AF file must install a C editor, because the file needs to be compiled to produce the final file. Another drawback is the compatibility of the audio files; the encoder supports only MP3 and, therefore the user has to convert his/her tracks to MP3. Nowadays this is not a big problem because there are lots of free software that converts audio files into MP3. But as the encoder developed in this thesis is a compressed file the resulting file has a low quality. This is more evident as the MP3 must be at 128 Kbps and the recommended bit rate is 320 Kbps.

The tracks, that a user would like to add to the file, have to be in the same folder and indicated (by the location) in the code. If for any reason a track is moved to another folder, the file will not be able to locate it. Also, the encoder does not support tracks stored online. All tracks must be in the computer. Due to lack of time, it neither supports metadata as image or text, which makes the application duller for players. These aspects can be easily incorporated in the future.

Another weakness of the developed file format is the difficulty to find music in multi-track format. For the time being, this encoder is intended for musicians or professionals who have access to the recordings of the song. This standard file hopefully would be embraced by major music labels such as EMI or SONY for the benefit of the consumer since it offers rich user experience. More important, they will supply with their large sound library thousands of songs in multi-track format. The new encoder will help in the popularization of the IM AF interactive file format to the general public. The intention is that this file format achieves enough popularity to be present in the recording studios as the standard format of the future.

## 6. Conclusions and further work

The task of creating an encoder is complex, especially if the file that requires this tool is new and very little has been written about it. This fact makes this challenge even more interesting and exciting. As many projects, especially in the engineering field, one needs to submit a product that meets some requirements, (in this case they are defined by the ISO) but one should have the freedom to develop it as considered best. The file created in this thesis is called Interactive Music Application Format (IM AF) and is described in ISO 23000-12. There, the different parts that this file should have are explained, like headers, space for samples and other data.

In this thesis we have first studied this standard and others for the subsequent development and implementation of a code that creates a file compatible with the decoder supplied by MPEG. The code is written in C and has two parts. One generates the framework of the file (IM\_AF Encoder.h) and the other fills this framework with the data needed and writes the file. The header includes different *boxes* that together form the framework of the file. Basically, a box limits (by the *size* variable) a section of memory in the file, which may contain information or other boxes. The way this storing procedure has been implemented in the code is by using the *struct* structures. They have the same behaviour when storing data and can hold another *struct* inside them. Moreover, since it is a binary file, the *struct* can be directly written in the file, thus facilitating the writing of the content. In this manner, it avoids the need for writing the file byte by byte.

The other section of the code (main.c) is in charge of introducing the information in the boxes and of writing them in the proper order. This part is the most complicated and elaborated one. It must handle several types of operations: from calculating the size of each box, to convert an MP3 into IM AF files. The conversion and processing of MP3 files has been a fundamental part in the success of the encoder. First, we have studied the standard that defines the MP3. Then, the MP3 files have been opened for analysing and understanding the real framework of the files; as each MP3 encoder varies the way the file is conformed. Once we knew how the file works, we read the information that is needed, and store it in the appropriate boxes. This process must be repeated for each track/instrument that one wants to insert in the file. I Consequently, most of the process inside the main code is part of a loop.

The management of the data has made very laborious the creation of the encoder. The reason why C was the chosen programming language is because it simplifies the programming tasks, as C is a complete but basic language that does not incorporate external libraries. These libraries in most cases are difficult to understand, as they require a high knowledge of the subject.

To complete the process of developing the IM AF encoder, three examples are presented. They prove the compatibility and the correct operation of the encoder showing to a new user how it looks in real world applications.

When I started this project, I knew the MPEG technologies from the lectures at the university. But I did not know how they worked in detail. I chose this topic because I felt attracted the first time I heard about it. IM AF technologies are a new music file format that will revolutionize the music industry. By creating this encoder, I present a tool that will help in this task, as it is an easy tool that many people will be able to use. This will surely contribute to the widespread use of the IM AF file format.

This thesis has helped me to improve my programming skills. Also I have become familiarized with a new and promising technology. However, there have been some obstacles during this process. The first one was to understand the file format and how I would implement it. Find the best suitable code and programming techniques were essential steps for the success in the development phase. The second obstacle was to eliminate artifacts produced by the wrong manipulation of samples. At the time of writing the *sample table box*, I wrote the data in the table a row after the correct position. This generated a noise that prevented the correct listening of the song. Correcting this and other errors was a very rewarding task as it has improved my knowledge in this field.

Despite that the encoder created in this thesis does its task properly; time limitations have prevented the implementation of some interesting additional tools. The ability to store images, like the cover of the album or pictures of the artist, and to insert synchronized text for karaoke, are two improvements that can be introduced in future versions of the encoder. Also, it would be very interesting to store the tracks online. This will help to create a social network where users could share instruments without the need to have physically the file.

On the other hand, IM AF is a new standard that creates an interactive file that allows manipulation, in terms of volumes, of the track. But it could also improve the file by adding more tools. The incorporation of an equalizer for each channel would be an interesting task for future developments. In addition to controlling the volume, the user would have the chance of manipulating the frequencies, thus converting the file to a more complete and professional tool. Another improvement would be to introduce a panorama for each channel, where the user would move the instrument in the 3D space, placing each track in a different position.

Concerning the technical part of the encoder there could also be some improvements. The programming language chosen in this project did not allowed to incorporate the encoder inside a mobile application, as most of the App work with C++ or Java. For this reason, translating the code to one of these languages will be useful for the future development of the encoder.

## References

- [1] ISO/IEC Std. 2010, *Information Technology – Multimedia application format (MPEG-A) – MPEG music player application format – Part 12: Interactive music application format*, ISO/IEC FDIS 23000-12.
- [2] ISO/IEC Std. 2006, *Information Technology – Multimedia application format (MPEG-A) –Part 2: MPEG music application format*, ISO/IEC 23000-2.
- [3] F. Gallot, O. Lagadec, M. Desainte-Catherine, S. Marchand. *iKlax: A New Musical Audio Format for Interactive Music*, Proc. ICMC (International Computer Music Conference) 2008, August 2008.
- [4] ISO/IEC Std. 2002, *Information Technology – Coding of Audio-Visual Objects – Part 1: Coding of audio-visual objects*, ISO/IEC 14496-1.
- [5] Microsoft Developers Network. Microsoft RIFF – WAVE. Available at: <http://oreilly.com/www/centers/gff/formats/micriff/>
- [6] ISO/IEC Std. 2006, *Information Technology – Coding of moving pictures and associated audio for digital media at up to about 1,5 Mbit/s – Part 3: Audio*. ISO/IEC 11172-3.
- [7] ISO/IEC Std. 1995, *Information Technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)*. 13818-7.
- [8] ISO/IEC Std. 2012, *Information Technology – JPEG 2000 image coding system – Part 12: ISO base media file format*. 15444-12.
- [9] ISO/IEC Std. 2002, *Information Technology – Coding of Audio-Visual Objects – Part 17: Streaming text format*, ISO/IEC 14496-17.
- [10] ISO/IEC Std. 2006, *Information Technology – Coding of moving pictures and associated audio for digital media at up to about 1,5 Mbit/s – Part 1: Systems*. ISO/IEC 11172-1.
- [11] ISO/IEC Std. 2007, *Information Technology – Generic coding of moving pictures and associated audio information – Part 1: Systems*. ISO/IEC 13818-1.
- [12] J. Watkinson. *MPEG-2 Chapter 4 Audio compression*. Oxford. Focal Press, 1999.
- [13] T. Painter, A. Spanias. *Perceptual coding of audio*. Arizona State University, 2000.
- [14] K. Brandenburg, H. Popp. *An introduction to MPEG Layer-3*. Fraunhofer Institut für Integrierte Schaltungen (IIS).
- [15] *MP3 File Structure*. Available at: <http://www.multiweb.cz/twoinches/mp3inside.htm>
- [16] F. Pereira. *MPEG-4: Why, what, how and when?* Instituto de Telecomunicações, Lisboa, 2002. Available at: <http://www.sciencedirect.com/science/article/pii/S0923596599000491>



- [17] F. Pereira, T. Ebrahimi. *The MPEG-4 Book*. New Jersey: IMSC Press, 2002.
- [18] L. A. Ludovico, *Key Concepts of the IEEE 1599 Standard*. Laboratorio di Informatica Musicale (LIM), Milano, 2008.
- [19] Draft 0.9.1a, *Interactive XMF: File Format Specification*. La Habra CA. February 18, 2008.
- [20] Larry the O., G. Sanger, B. Starr. *Group Report: Towards Interactive XMF*. Project BAR-B-Q 2001. The Sixth Annual Interactive Music Conference. Available at: <http://www.projectbarbq.com/bbq01/bbq01r5.htm>
- [21] ISO/IEC Std. 2002, *Information Technology – Coding of Audio-Visual Objects – Part 12: ISO base media file format*, ISO/IEC 14496-12.
- [22] A. Eleftheriadis. *The MPEG-4 System and Description aguages: From Practice To Theory*. Columbia University, Ney York.
- [23] ISO Std. 2004, *Data elements and interchange formats – Information interchange – Representation of dates and times*, ISO 8601.
- [24] ISO Std. *Codes for the representation of names of languages – Part 2*. ISO 639-2.
- [25] ISO Std. 1998, *Information Technology – Generic coding of audio-visual objects – Part 1: Systems section*. ISO/IEC 14496-1.
- [26] *Mp4 Explorer*. Open Source Software created by CodePlex. Available at: <http://mp4explorer.codeplex.com/>
- [27] *MP4 Browser* by MiraVid. Available at: <http://download.cnet.com/MiraVid-MP4-Browser>
- [28] *The 'Mixing Secrets'*, Free multitrack download library. Available at: <http://www.cambridge-mt.com/ms-mtk.htm#Acoustic>
- [29] *Easy Mercurial*. Developed in the Center of Digital Music of Queen Mary.
- [30] R. Stewart, P. Kudumakis, M. Sandler. *Interactive Music Applications and Standards in Exploring Music Contents*. Lecture Notes in Computer Science, Vol. 6684, Springer Berlin Heidelberg, Editors: S. Ystad, M. Aramaki, R. Kronland-Martinet & K. Jensen. Aug. 2011
- [31] I. Jang, P. Kudumakis, M. Sandler, K. Kang. *The MPEG Interactive Music Application Format Standard*, IEEE Signal Processing Magazine, pp. 150-154, Vol. 28, Issue 1, Jan. 2011.
- [32] E. Onate, P.Kudumakis. *Development of an IM AF encoder*. Queen Mary, London, 2012.
- [33] The code with the IM AF examples can be downloaded from: <https://code.soundsoftware.ac.uk/hg/enc-imaf>

## Appendix.1 – Paper: Development of an IM AF encoder

# Development of an IM AF encoder

Eugenio Onate, Panos Kudumakis  
School of Electrical Engineering and Computer Science  
Queen Mary University of London  
Email: eo301@eecs.qmul.ac.uk

**Abstract**—In this work an encoder able to create a new interactive file following the so-called Interactive Music Application Format (IM AF), has been developed. A file created with the IM AF allows users more control over the song. More specifically, it permits to vary the volume of each instrument or choose a predefined preset. The encoder has been programmed in C following the standard defined by MPEG. The paper describes first the previous file formats which have been the basis for the development of the IM AF file. Then the main features and the structure of the new encoder are detailed. The performance of the encoder has been validated by creating three IM AF files with special music features. These conformance files have been successfully tested in the IM AF player provided by the MPEG group.

### I. INTRODUCTION

The music market is strangled with decreasing sales. Customers need new products that attract their attention. The time has come to innovate and create a new file format that will change the way people listen the music. Formats like mp3 were a good revulsive for industry, but have been here since the 90s; today it is all about interactivity between user and technology. Nowadays technologies have led people to live in a society where everyone is connected among themselves and sharing all kind of information.

The music industry has been reluctant to let people interact with the songs, basically because the technology was not ready. Now it is the perfect time to change the concept of the listener, making him/her participate of the musical experience. The Moving Picture Experts Group (MPEG) defined a new file format called Interactive Music Application Format (IM AF) [1] as part of the Multimedia application format (MPEG-A) [2]. IM AF is a versatile file format standard for mixing different types of multimedia data (music, images and text). It allows users to modify the volume of each instrument separately or change the mixing style according to some presets predefined by the producer.

The process for creating standardized files like MPEG-1 Layer III (.mp3) or IM AF (.ima) is split in two sections: encoder and decoder. In the encoder section, the file is created following the appropriate standard; parts like headers, track information and samples data are put together inside a binary file. The decoder is responsible to read and understand the file so that it will be able to reproduce it.

When a new file is defined, the file decoder is made publicly available through ISO/IEC MPEG to let companies design their own encoder. In this manner, they ensure compatibility of the file no matter how it was created. Since MPEG defined

the standard for IM AF in 2010 no one has publicly released an implementation of the encoder yet, although commercial services exist.

In this work an encoder for IM AF files has been designed, implemented and tested following the ISO 14496 Part 12: ISO base media file format and ISO 23000 Part 12: Interactive music application format. The new IM AF file encoder can support a maximum of 16 simultaneously audio tracks with a sampling frequency of 44.1kHz at 16 bits per sample. In this version, individual music-tracks must be encoded in MP3. Also, it is able to add different mixing presets and rules. Presets are predefined values of the volume of each instrument that allow the producer to create different versions of the track. Also users can exchange and share their own mixtures. Rules are limitations imposed by the creator of the song, usually producers, to avoid users destroy the essence of the song. The encoder has been programmed in C and has a simple command line user interface for introducing the information required by the program.

### II. BACKGROUND RESEARCH

In this section we analyze some previous standards that have contributed to create the IM AF file format. We also present similar applications based on interactivity files.

#### A. MPEG-1 Layer III

MPEG-1 Layer III, most commonly known as MP3 [3], was released in 1991 and soon become the most used tool for Internet and audio delivery. It became very popular due to its big impact on the music industry, offering good sound quality with low bit-rate. MPEG-1 works on different sampling frequencies and supports variable compression ratio. For Layer-III the standard determines a variety of bit-rates from 8 Kbit/s to 320 Kbit/s. The most common ones are 192 Kbit/s and 320 Kbit/s as they provide transparent quality. Bit-rate can change from frame to frame allowing higher bit-rates for more complex parts of the song, while less space is needed for less complex ones.

Audio encoders may have different architectures depending on the model. MPEG standard encoders have no restrictions for their design. However, all files produced have to be compatible with the corresponding decoder. A MP3 file is split into small blocks or frames where data is stored. Each frame has 1152 samples per frame [4]. The standard sampling frequency is 44.1kHz, which means that the duration of one frame is  $1152/44100 = 0.026$  sec. Knowing the bit-rate it is possible

to calculate the size in Bytes of one frame [4]; if the bit-rate is 128kbps the size is 417 Bytes, while if it is 192kbps the size is 626 Bytes. At beginning and end, MP3 files may have ID3 TAGs. Those are metadata containers with information related to the song.

[ID3 TAG] Frame1 Frame2 Frame3 ... FrameN [ID3 TAG]

Each frame is structured in the same way. It has a 32-bit header followed by the samples. The header contains information associated to technical specifications such as MPEG version, Layer, Bit-rate and sampling frequency. The decoder uses this information to decode the file properly.

#### A. MPEG-4

MPEG-4 was an evolution of MPEG-2. Unlike its predecessor, improving the compression method was not the main task. The main difference with previous audiovisual coding standards is the object-based representation model that helped consolidate MPEG-4. Objects can be audio objects like multi-channel audio content or speech, or video and movies. Transmitting several audio objects with multiple tools creates an audio composition system or soundtrack. This ability furnishes MPEG-4 with important superiority in quality and flexibility versus its predecessors.

The file format defined by MPEG-4 is called MP4; it is described in ISO 14496 Part 12 Base Media File Format. It is an important part of the IM AF file as explained in a next section. It is intended to accommodate multi-media information in a versatile arrangement that allows ease manipulation, correction and display of the media.

#### B. Previous interactive music systems

IM AF was not the first file format to include interactive control over the media. In this section we briefly describe three examples of earlier formats: IEEE 1599 [5], iXMF [6], and iKlax [7].

IEEE 1599 combines in a single file music and XML symbols, as graphical representation of the music or performance indications. This information is integrated and synchronized within the same framework and can be accessible individually or as a whole. An exhaustive description of music must integrate different types of information. IEEE 1599 has developed a new XML encoding system to allocate this heterogeneous information in a single file. XML organizes the information in six different layers [5]

- General - It stores information about the piece.
- Logic - Coherent representation of score symbols.
- Structural - Classifying musical objects and the connection.
- Notational - Visual representation of the score.
- Performance - Computer-based description of musical representation.
- Audio - Digital audio recording.

iXMF or interactive eXtensible Music Format is a file format created by the video games industry to provide a standard

structured audio file format that supports cross-platform interchange of advanced interactive audio tracks. An iXMF file includes in a single file all the information needed to perform the audio track as the artist intended. The same information defines the structure of the file. iXMF uses a structure such that an event can be triggered at a particular instant. The selected event can activate a wide range of activities, such as the playing of an audio file or the execution of a specific code.

iKlax technology was developed by iKlax Media company and LaBRI in France. iKlax proposes a file format with separated tracks and interactivity to manipulate the music piece. The project includes a music player and a music editor. iKlax format groups all the tracks of a song and related metadata in a single file. It has two levels of interactivity; the first level allows the selection of the track, where listeners can choose the track that they want to listen. The second level is the mixing; here listeners can modify the level of each track.



Fig 1. iKlax Player

#### I. DEVELOPMENT OF THE IM AF ENCODER

In this section we present the IM AF encoder developed in this work. Also, we will analyze the different sections of the file that are used in the new encoder. IM AF uses parts of the MPEG-4 standard as well as new parts designed specifically for this new interactive file.

##### A. Interactive music service

To ensure interoperability between different multimedia content, the International Standard Organization (ISO) presented in 2010 IM AF. IM AF specifies how to connect multiple tracks with related information under a well-defined structure that facilitates the manipulation of interactive music content. This content includes audio tracks, preset data and rules. Audio tracks represent the instruments of the song, they can be a single instrument or a group. Preset data is a predefined information related to the volume of each track and allows users to create different versions of the song. Those values cannot be changed after the file is created. Hence, it can be useful for producers to present the same track from different points of view. Finally, IM AF files introduce rules to avoid users destroy the initial intention of the author. The created files are reproduced by an interactive music player

as shown in Figures 2 and 3. Users have two different options to listen the song: preset-mix mode and user-mix mode. In the preset mode the user choses one of the pre-defined presets in the IM AF file, and then the tracks change the volume according to the preset values. In the user mode, the user selects/de-selects the tracks and controls their volume. All the actions performed by the user need to be compatible with the rules; otherwise the actions will not be carried out.

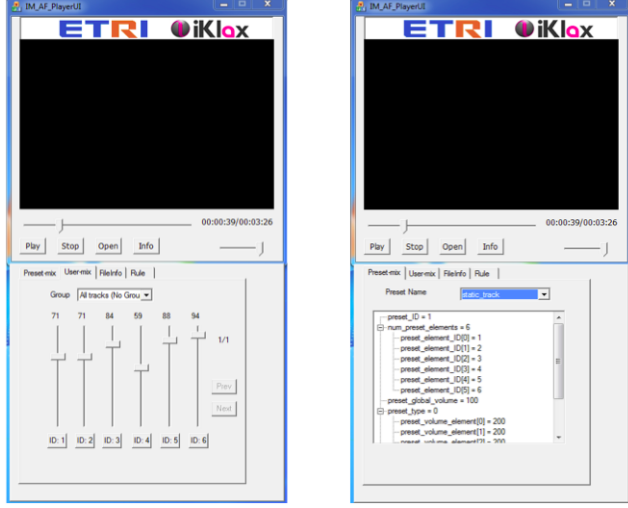


Fig 2. IM AF player. User-mix mode (left) and preset mode (right).

#### A. Structure of the IM AF file

The encoder is responsible for creating the IM AF file. For doing that it follows the standard defined in ISO 23000-12. The framework of this file is based on the MPEG-4 ISO based Media File Format standard; IM AF has introduced some improvements to enable interactive control.

IM AF files consist of a series of boxes that include all data. There are two different types of boxes; those that may contain other boxes inside them and others that just contain data (called *FullBoxes*). All boxes start with the header which defines the *size* and *type*. *FullBoxes* incorporates in the header the *version* and *flag* information. The *size* defines the total size of the box, including data and, if necessary, other boxes. It has a size of 32 bits (unsigned integer), but if the data stored in the box is bigger than that it can use 64 bits. *Type* is the identification of the box, and each box has its own type, i.e. *ftyp*, *moov*, *mdia*. The *version* is an integer (8 bits) and specifies the version of the box, and the *flag* is a 24-bit integer and its use depends on each box.

A complete list of the boxes that form the structure is presented below:

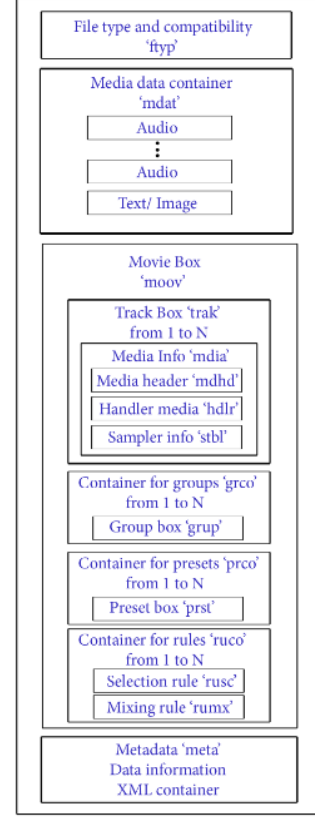


Fig 3. Boxes inside the IM AF file

We will not explain each box individually, further information and description of each box can be found in [1], [8], and [9].

#### I. IMPLEMENTATION

The encoder that creates the IM AF file has been implemented in C. It does not use any external library, ensuring that it will work easily in any computer. The program has two parts: the main program and the IM AF header. The main program includes the functions to create the file and the header defines the structure of the boxes. There is also a basic command line tool to let users introduce values such as the number of tracks and their names. The program has a static structure; meaning that the maximum number of tracks, presets and rules are defined by a global variable.

Right at the beginning, the encoder asks the user to introduce the number of tracks that the IM AF file will incorporate. Then, the user writes the names of all the tracks to be stored in the file. The encoder detects if the track exists or not. If it does not, the program will ask again for the name. When the user has finished to introduce all the tracks, the encoder creates the binary file that will store all the information. The first function called in by the program is the *filetypebox*. It specifies the different types supported by the encoder.

Once the type is clear, the encoder converts the audio files (MP3) into MP4 files. The first step is to extract the samples

of the audio file and store them into the *media data container*. The idea is to find the beginning of the first frame and read the data until it reaches the end of the file. The intention is to avoid the ID3 header, as it does not contain useful information. The second step is to extract the sample information from the frames of the MP3 and write it in the corresponding sample box inside the *Track Box*. In the *sample size box* stores the duration and size of each frame, as well as the total number of frames (Figure 4). In this case the search for information is more precise than in the *media data container*. Rather than reading the whole file, it localizes the beginning and end of each frame, extracting the information of one frame at a time. This information is reused by the *time to sample*, *sample to chunk* and *chunk offset* boxes.

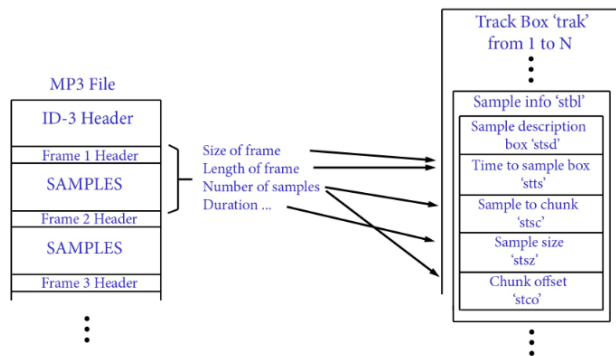


Fig 4. Sample data information

The *preset container* is next implemented after the track box. It creates a static preset with a fixed value. First, it defines the number of presets that there will be in the file. Then, it assigns the volume of each track; initially takes the value of 100. Changing this value will modify the volume of each track in the specific preset. The next step is to create the rules. The rule container creates two rules: one selection rule and one mixing rule. Finally, the movie box is created together with the movie header box. Here the encoder puts together the sizes of all boxes within the movie box. Doing that at the end of the code ensures that the total size of the movie box will match the sum of each one.

#### A. Programs that analyze the IM AF file

In the process of creating the IM AF file we have used various programs that analyze the structure of the file. Specifically programs that show the boxes of the MPEG-4 file and programs that work for IM AF too. The creation of a file like the one created here is full of little parameters that can lead to ill-function of the file. The order of the boxes is very important, so it is the size of each box. These programs have been an important part of the success of the encoder. The *MP4 Browser* [10] is a free software for MS Windows. It gives a very clear idea of how the IM AF file is structured. It details the content, *type* and *size* of all *boxes*, except the ones used only in IM AF, like the *preset* and *rules boxes*. This software

was created to work with ISO 14496 files (MPEG-4), so it does not support the new features of IM AF. Without this type of programs the task of building an IM AF encoder would be more difficult.

## I. RESULTS

In this section we analyze the characteristics of the files created by the encoder. When the user wants to create a new file, he/she introduces the information through a command line interface. Before presenting the conformance files, we will explain the steps for doing that task. First, the encoder welcomes the user and asks for the number of tracks to be included in the IM AF file.

```
Welcome to the IM_AF encoder
This program will allow you to create an IM_AF file.
How many tracks there will be in your IMAF file?
3
```

Fig 5. Creation of file - Step 1

In this case the user has selected 3 tracks. The maximum number of tracks that the IM AF file supports is limited by the larger brand in the *filetypebox*. After specifying the number of tracks, the next step for the user is to enter the name of each track. The audio files must be in the same folder as is declared in the code and the names have to be written together with their extension, as showed below.

```
How many tracks there will be in your IMAF file?
3
Name of the track number: 1
bass.mp3
Name of the track number: 2
drums.mp3
Name of the track number: 3
synth.mp3
```

Fig 6. Creation of file - Step 2

When the user has finished entering all the names, the encoder creates the preset defined previously in the code. To provide clarity on what is being written in the file, the encoder prints on the screen the characteristics of the preset.

```
Presets:
Static track volume preset: invariant volume related to each track
-----
Preset number 1: static_track
Enter volume for bass.mp3 = 120
Enter volume for drums.mp3 = 100
Enter volume for synth.mp3 = 80
```

Fig 7. Creation of file - Step 3

The last step is to write the rules into the file. They are also written directly in the code rather than using the command line.

**Rules:**  
**Rule 1: Not mute for channel 3**  
**Rule 2: Upper rule between channel 1 and 2**  
**File is created successfully, and ready to use!**

Fig 8. Creation of file - Step 4

After that, the encoder writes all the information into the file. Hence, the only thing that remains to be done is to inform to the user that the file is created. A file with extension IM AF ready to play in the IM AF player has been created.

#### A. Conformance files

Three different files have been implemented in order to validate the performance of the new encoder. Each time a new IM AF file is presented we supply various examples to prove the efficiency of the encoder. For that purpose, we have selected three different multi-track songs from [11] with different styles. Each song has different configurations of the encoder parameters. Below it presents one conformance file, the rest of the file can be found in [8].

Example 1	
Name:	Example_Acoustic.ima
Format:	MP3
#Instruments	6
Presets	Yes
Rules	Yes (2)

Type/N° of instrument	Value of Static Preset	Selection Rule: Not mute	Mixing Rule: Equivalence
ID 1 - Drums	40	-	Element involved
ID 2 - Bass	80	-	Key element
ID 3 - Voice	120	Element involved	-
ID 4 - Guitar	180	-	-
ID 5 - Clarinet	200	-	-
ID 6 - Accordion	240	-	-

Fig 9. Conformance file 1

This example has six instruments, two rules and one static preset. The selection rule is Not mute and the element that will be always in active state is number 3: the voice. The rest of the instruments can be mute. The mixing rule is Equivalence, meaning that two elements will have the same volume during the whole piece of music. In this case the elements involved are the drums and the bass. At no time one element will sound louder than the other. The preset defines a static volume for the instruments; the secondary elements like the clarinet or accordion get more presence and the important ones like the drums or voice reduce the level.

#### I. CONCLUDING REMARKS

In this work an IM AF encoder has been implemented that creates a new interactive file. It is intended for all users, from a beginner that does not know much about programming to a professional engineer who wants to personalize the encoder to suit his/her needs. The beginner will find the encoder useful and intuitive as he/she does not need to modify the code for generating an IM AF file. There is a command line interface that guides newcomers through the creation of the file. This

interface allows users to enter the instruments in the file. The customization of the encoder by a user is also a relatively simple task.

On the other hand, an experimented user will find the encoder sufficiently robust tool for his/her professional work. Moreover, in case a user would want to introduce new improvements in the file, the program created will represent the framework of his/her work and will be flexible enough to support the changes.

This encoder has been created in a relatively short period of time. Thus, there are some aspects that could have been done differently. The command line interface is limited; the user can introduce information related to the number and name of the tracks. Also the presets and rules information must be written directly in the code, hence making it troublesome for someone who does not know C. Life would be more comfortable for a user if the interface was an application by itself, rather than being part of the C compiler. The encoder supports only MP3 files. Nowadays this is not a problem as there are lots of free software that converts audio files into MP3.

A weakness for the immediate application of the IM AF file format developed is the difficulty to find music in multi-track format. For the time being, this encoder is intended for musicians or professionals who have access to the recordings of the song. This standard file hopefully would soon be embraced by major music labels such as EMI or SONY via their huge sound library. In this manner, the new encoder will help in the popularization of the IM AF interactive file format to the general public. The intention is that this file format achieves enough popularity to be present in the recording studios as the standard format of the future.

Future extensions of the decoder will include the ability to store images, like the cover of the album or pictures of the artist, and the possibility to insert synchronized text for karaoke. Also, it would be interesting to store the tracks online. This will help to create a social network where users could share instruments without the need to have physically the file. We note that IM AF is a new standard that creates an interactive file that allows the manipulation of the track volume. This file can be improved by adding more tools such as an equalizer for each channel. In this manner, users would have the chance of manipulating the frequencies in addition to controlling the volume, thus converting the file into a more complete and professional tool. Another improvement would be to introduce a panorama for each channel, where users could move the instrument in the 3D space, placing each track in a different position. We finally point out that the programming language chosen did not allow to incorporate the encoder inside a mobile application. Thus, translating the code to C++ or Java will be useful for the future development and implementation of the encoder in Apps.

#### REFERENCES

- [1] ISO/IEC Std. 2010, *Information Technology Multimedia application format (MPEG-A) MPEG music player application format Part 12: Interactive music application format*, ISO/IEC FDIS 23000-12.

- [2] ISO/IEC Std. 2006, *Information Technology Multimedia application format (MPEG-A) Part 2: MPEG music application format*, ISO/IEC 23000-2.
- [3] ISO/IEC Std. 2006, *Information Technology Coding of moving pictures and associated audio for digital media at up to about 1,5 Mbit/s Part 3: Audio*, ISO/IEC 11172-3.
- [4] *MP3 File Structure* Available at:  
<http://www.multiweb.cz/twoinches/mp3inside.htm>
- [5] L. A. Ludovico, *Key Concepts of the IEEE 1599 Standard*, Laboratorio di Informatica Musicale (LIM), Milano, 2008.
- [6] Draft 0.9.1a, *Interactive XMF: File Format Specification*. La Habra CA, February 18, 2008.
- [7] F. Gallot, O. Lagadec, M. Desainte-Catherine, S. Marchand, *iKlax: A New Musical Audio Format for Interactive Music*. Proc. ICMC (International Computer Music Conference) 2008, August 2008.
- [8] E. Onate, *Development an IM AF encoder*. MSc Digital Music Processing Final Report, Queen Mary, 2012.
- [9] ISO/IEC Std. 2002, *Information Technology Coding of Audio-Visual Objects Part 12: ISO base media file format*, ISO/IEC 14496-12.
- [10] *MP4 Browser* by MiraVid. Available at:  
<http://download.cnet.com/MiraVid-MP4-Browser>
- [11] *The Mixing Secrets*, Free multitrack download library. Available at:  
<http://www.cambridge-mt.com/ms-mtk.htm#Acoustic>



## Aendix.2 - Listening of the IM AF encoder program

### Main.c

```
// main.c
// IM_AM Encoder
//
// Created by Eugenio Oñate Hospital on 14/06/12.
// Copyright (c) 2012 QM. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "IM_AF Encoder.h"

/*Prototype*/

void filetypebx(FileTypeBox *ftyp);
int mdatbox(MediaDataBox *mdat, int, FILE *imf, FILE *song, int);
void moovheaderbox(MovieBox *moov, int, int, int, int, int, int);
int trackstructure(MovieBox *moov, int, int, int, int, char name[20]);
int samplecontainer(MovieBox *moov, int, int, char name[20]);
int sampledescription(MovieBox *moov, int);
int presetcontainer(MovieBox *moov, int, nametrack namet);
int rulescontainer(MovieBox *moov);
void writemoovbox(MovieBox moov, int numtrack, int totaltracks, FILE *imf);
int readTrack(MovieBox *moov, int, char name[20]);
int byterevers(int);
```

### Main function

```
int main ()
{
    //variables
    FileTypeBox ftyp;
    MediaDataBox mdat;
    MovieBox moov;
    // MetaBox meta;
    nametrack namet;

    FILE *imf;
    int numtrack, totaltracks, sizemdat, durationTrack;

    /* Obtain current time as seconds elapsed since the Epoch. */
    time_t clock = time(NULL);

    printf("\nWelcome to the IM_AF encoder\n");
    printf("This program will allow you to create an IM_AF file.\n");
    printf("How many tracks there will be in your IMAF file?\n");
    scanf("%d", &totaltracks);
    fflush(stdin);
    while (totaltracks > maxtracks) {
        printf("Sorry, for this version the number maximum ot tracks is %d\n", maxtracks);
        printf("How many tracks there will be in your IMAF file:\n");
        scanf("%d", &totaltracks);
    }

    //Create the file
    imf = fopen ("/Users/eugin/Desktop/IM_AF Encoder/IM_AM Encoder/Example_Rock.ima", "wb");
    if (imf == NULL) {
        printf("Error opening input file\n");
        system("pause");
        exit(1);
    }

    //Define the File Type Box
    filetypebx(&ftyp);
```



```

fwrite(&ftyp, sizeof(FileTypeBox),1, imf);

//Media Data Box – Contains the audio
FILE *song;
char nametrack[20];
//Specify the path directory where there are the songs.
//If change folder, change the path here (3 times) and in readTrack function!!!
char pathdir[60] = "/Users/eugin/Desktop/IM_AF Encoder/Rock/";
int numtr, ex = 0;
for (numtr=0; numtr<totaltracks; numtr++) {

    printf("Name of the track number: %d\n", numtr+1);
    fflush(stdin);
    scanf("%s", nametrack);
    strcpy(pathdir, "/Users/eugin/Desktop/IM_AF Encoder/Rock/");
    strcat(pathdir, nametrack);
    ex = 0;
    //Check if the track exist and then open it.
    while (ex == 0){
        song = fopen(pathdir, "rb");
        if((song)==NULL) {
            printf("Name does not exist. Try again:\n");
            fflush(stdin);
            scanf("%s", nametrack);
            strcpy(pathdir, "/Users/eugin/Desktop/IM_AF Encoder/Rock/");
            strcat(pathdir, nametrack);
        }else{
            ex = 1;
        }
    }
    strcpy(namet[numtr].title, nametrack);

    //Extract the samples from the audio file
    sizemdat = mdatabox(&mdat, totaltracks, imf, song, numtr);

    //Close the audio file
    fclose(song);
}

//For each track write track information
u32 sizeTRAK = 0;
char name[20];
durationTrack = (sizemdat*8)/128;

for (numtrack = 0; numtrack < totaltracks; numtrack++) {
    strcpy(name,namet[numtrack].title);
    sizeTRAK = trackstructure(&moov, numtrack, clock, durationTrack,sizemdat,
name)+ sizeTRAK;
}

//Presets
u32 sizePRCO;
sizePRCO = presetcontainer(&moov, totaltracks, namet); // Creates the preset,
returns the size of the box.

//Rules
u32 sizeRUCO;
sizeRUCO = rulescontainer(&moov); // Creates the rules, returns the size of the
box.

//Movie Header – Overall declarations
moovheaderbox(&moov, clock, sizeTRAK, sizePRCO, totaltracks, durationTrack,
sizeRUCO);

//Writes the movie box into the file
writemoovbox(moov,numtrack, totaltracks, imf);

//Close File
fclose(imf);

printf("\nFile is created successfully, and ready to use!\n");

return 0;
}

```

## File Type box

```
void filetypebox(FileTypeBox *ftyp){
    int swap;

    swap = byterevers (24);
    ftyp->size = swap;
    swap = byterevers ('ftyp');
    ftyp->type = swap;
    swap = byterevers ('im03');
    ftyp->major_brand = swap;
    ftyp->minor_version = 0;
    swap = byterevers ('im03');
    ftyp->compatible_brands[0] = swap;
    swap = byterevers ('isom');
    ftyp->compatible_brands[1] = swap;
}
```

## Media data box

```
int mdatbox(MediaDataBox *mdat, int totaltracks, FILE *imf, FILE *song, int numtr){

    int d, cnt, j, find = 0;
    int dat = 0, dat1 = 0, dat2 = 0, dat3 = 0;
    u32 size = 0, swap, sizeMDAT = 0;
    //Positionate the pointer at the end of the file to know the size of it
    fseek(song, 0, SEEK_END);
    size = ftell(song);
    //Positionate the pointer at first
    fseek(song, 0, SEEK_SET);
    d=0;
    cnt = 0;
    //Find the header of the first frame (the beginning), when find it d=1 and jump
    out the loop.
    // The header is 32 bytes. We find in groups of 8 bytes
    // Contemplate all possible options of headers
    while (d == 0) {
        find = 0;
        fread(&dat, sizeof(unsigned char), 1, song);
        cnt++;

        if (dat == 0xFF) {
            cnt++;
            position of the pointer.
            fread(&dat1, sizeof(unsigned char), 1, song);
            cnt++;
            fread(&dat2, sizeof(unsigned char), 1, song);
            cnt++;
            fread(&dat3, sizeof(unsigned char), 1, song);
            if (dat1 == 0xFB && dat2 == 146 && dat3 == 64 ) {
                find = 1;
                is found
                d=1;
                loop
            }
            if (dat1 == 0xFB && dat2 == 146 && dat3 == 96 ) {
                d=1;
                find = 1;
            }
            if (dat1 == 0xFB && dat2 == 144 && dat3 == 64 ) {
                find = 1;
                d=1;
            }
            if (dat1 == 0xFB && dat2 == 144 && dat3 == 96 ) {
                find = 1;
                d=1;
            }
            if (dat1 == 0xFB && dat2 == 146 && dat3 == 100 ) {
                d=1;
                find = 1;
            }
            if (dat1 == 0xFB && dat2 == 144 && dat3 == 100 ) {
                find = 1;
                d=1;
            }
        }
    }
}
```

```

        if (dat1 == 0xFA && dat2 == 146 && dat3 == 64 ) {
            find = 1;
            d=1;
        }
        if (dat1 == 0xFA && dat2 == 146 && dat3 == 96 ) {
            d=1;
            find = 1;
        }
        if (dat1 == 0xFA && dat2 == 144 && dat3 == 64 ) {
            find = 1;
            d=1;
        }
        if (dat1 == 0xFA && dat2 == 144 && dat3 == 96 ) {
            find = 1;
            d=1;
        }
        if (dat1 == 0xFA && dat2 == 146 && dat3 == 100 ) {
            d=1;
            find = 1;
        }
        if (dat1 == 0xFA && dat2 == 144 && dat3 == 100 ) {
            find = 1;
            d=1;
        }
        if (find == 0) {
            fseek(song, -3, SEEK_CUR);
            cnt = cnt - 3;
        }
    }
    if (cnt == size) {
        d = 1;
    }
}
size = size - (cnt - 4);          // Calculate the size of the samples. size = pos.
end of file - pos. first header.
if (numtr == 0) {
    sizeMDAT = size*totaltracks + 8;    // size of the whole media box
    swap = byterevers(sizeMDAT);
    fwrite(&swap, sizeof(u32), 1, imf);
    swap = byterevers('mdat');
    mdat->type = swap;
    fwrite(&mdat->type, sizeof(u32), 1, imf);
}
fseek(song, cnt - 4, SEEK_SET);
for (j=0; j<size; j++) {           //read all the samples of one track and
writes them in the IM AF file
    fread(&mdat->data, sizeof(char), 1, song);
    fwrite(&mdat->data, sizeof(char), 1, imf);
}
fclose(song);

return size;
}

```

## Sample Container

```

int samplecontainer(MovieBox *moov, int numtrack, int sizemdat, char name[20]){

    u32 sizeSTSD, sizeSTSZ, swap, num_samples, dat=0;

    //Sample Description Box//
    sizeSTSD = sampledescription(moov, numtrack);

    //Sample size box//
    swap = byterevers('stsz');
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleSizeBox.type = swap;
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleSizeBox.version = 0;
    //Read Track: Frame size and Decoder Times
    num_samples = readTrack(moov, numtrack, name);
    sizeSTSZ = num_samples*4 + 20;
    swap = byterevers(sizeSTSZ);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleSizeBox.size = swap;
}

```

```

//Time To Sample Box//
u32 sizetime, sizeSTTS;
sizetime = byterevers(moov->TrackBox[numtrack].MediaBox.MediaInformationBox.
    SampleTableBox.TimeToSampleBox.entry_count);
sizeSTTS = 16 + sizetime*4*2;
swap = byterevers(sizeSTTS);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.size = swap;
swap = byterevers('stts');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.version = 0;

//Sample To Chunk//
u32 sizeSTSC = 28;
swap = byterevers(sizeSTSC);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.size = swap;
swap = byterevers('stsc');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.version = 0;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.entry_count = swap;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.first_chunk = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.samples_per_chunk = moov->
>TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.SampleSizeBox.sample_c
ount;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleToChunk.sample_description_index = swap;

//Chunk Offset Box//
u32 sizeSTCO = 20;
swap = byterevers(sizeSTCO);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    ChunkOffsetBox.size = swap;
swap = byterevers('stco');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    ChunkOffsetBox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    ChunkOffsetBox.version = 0;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    ChunkOffsetBox.entry_count = swap;
dat = 32 + sizemdat*numtrack;
swap = byterevers(dat);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    ChunkOffsetBox.chunk_offset[numtrack] = swap;

//Sample Table Box //
u32 sizeSTBL = 8 + sizeSTSD + sizeSTSZ + sizeSTSC + sizeSTCO + sizeSTTS;
swap = byterevers(sizeSTBL);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.size = swap;
swap = byterevers('stbl');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.type = swap;

return sizeSTBL;
}

```

## Sample description

```

int sampledescription(MovieBox *moov, int numtrack){
    u32 swap, sizeESD = 35;
    swap = byterevers(sizeESD);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
        SampleDescriptionBox.AudioSampleEntry.ESbox.size = swap;
    swap = byterevers('esds');
}

```

```

moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.version = 0;

//ES Descriptor//
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.tag = 3;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.length = 21;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.ES_ID = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.mix = 0;

//Decoder config descriptor//
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.tag = 4;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.length = 13;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.objectProfileInd = 0x6B;
swap = byterevers(0x150036B0);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.mix = swap;
swap = byterevers(128);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.maxBitRate = swap;
swap = byterevers(128);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
DecoderConfigDescriptor.avgBitRate = swap;

//SLConfig Descriptor//
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
SLConfigDescriptor.tag = 6;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
SLConfigDescriptor.length = 1;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.ESbox.ES_Descriptor.
SLConfigDescriptor.pdefined = 2;

//Audio Sample Entry//
u32 sizeMP4a = 36 + sizeESD;
swap = byterevers(sizeMP4a);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.size = swap;
swap = byterevers('mp4a');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[0] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[1] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[2] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[3] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[4] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved[5] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.data_reference_index = 256;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved2[0] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.

```

```

SampleDescriptionBox.AudioSampleEntry.reserved2[1] = 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.channelcount = 512;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.samplesize = 4096; // 16 bits
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.reserved3 = 0;
swap = 44100 << 16;
swap = byterevers(swap);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.AudioSampleEntry.samplerate = swap;

//Sample description box //
u32 sizeSTSD = 16 + sizeMP4a;
swap = byterevers(sizeSTSD);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.size = swap;
swap = byterevers('stsd');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.version = 0;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleDescriptionBox.entry_count = swap;

return sizeSTSD;
}

```

## Read Track

```

int readTrack (MovieBox *moov, int numtrack, char name[20]){
    FILE *song;
    int d=0, cnt = 0, i=0, j=0, cnt2 = 0, find = 0, swap, num_entr = 0;
    int dat = 0, dat1 = 0, dat2 = 0, dat3 = 0, num_frame = 0, end =0, pos = 0;
    u32 size[9000];
    //Change path directory here
    char pathdir[60] = "/Users/eugin/Desktop/IM_AF Encoder/Rock/";
    strcat(pathdir, name);
    //Open the audio file with the name introduced by the user
    song = fopen (pathdir,"rb");
    if (song == NULL) {
        printf("Error opening input file\n");
        system("pause");
        exit(1);
    }
    //Calculate the size of the track
    fseek(song, 0, SEEK_END);
    end = ftell(song);
    fseek(song, 0, SEEK_SET);
    d=0, i=0;
    //Search for each frame one by one, and extratcs the information
    while (d == 0) {
        find = 0;
        fread(&dat, sizeof(unsigned char), 1, song);
        cnt++;

        if (dat == 0xFF) {
            cnt++;
            fread(&dat1, sizeof(unsigned char), 1, song);
            cnt++;
            fread(&dat2, sizeof(unsigned char), 1, song);
            cnt++;
            fread(&dat3, sizeof(unsigned char), 1, song);
            if (dat1 == 0xFB && dat2 == 146 && dat3 == 64 ) {
                pos = cnt - 4; //Pos of the beginning of the
                size[num_frame] = pos - cnt2; //Size of one frame
                cnt2 = pos; //Pos of the next frame
                find = 1;
                num_frame ++; //Number of frames
            }
        }
        if (dat1 == 0xFB && dat2 == 146 && dat3 == 96 ) {
            pos = cnt - 4;

```

```

        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFB && dat2 == 144 && dat3 == 64 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFB && dat2 == 144 && dat3 == 96 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFB && dat2 == 146 && dat3 == 100 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFB && dat2 == 144 && dat3 == 100 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 146 && dat3 == 64 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 146 && dat3 == 96 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 144 && dat3 == 64 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 144 && dat3 == 96 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 146 && dat3 == 100 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }
    if (dat1 == 0xFA && dat2 == 144 && dat3 == 100 ) {
        pos = cnt - 4;
        size[num_frame] = pos - cnt2;
        cnt2 = pos;
        find = 1;
        num_frame ++;
    }

```

```

    }
    if (find == 0) { //In case it does not find the header.
        //It keeps reading next data without jump any position
        fseek(song, -3, SEEK_CUR);
        cnt = cnt - 3;
    }
}

if (cnt == end) {
    pos = cnt;
    size[num_frame] = pos - cnt2;
    d = 1;
}
}

//Save Samples size//
swap = byterevers(num_frame);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleSizeBox.sample_count = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
SampleSizeBox.sample_size = 0;

for (i=0; i< num_frame; i++) {
    swap = byterevers(size[i+1]);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    SampleSizeBox.entry_size[i] = swap;
}

//Save Decoding Times//
//Writes manually the duration of each frame.
//Follows the following structure:
// 7 frames of 26 ms
// 1 frame of 27 ms
// ...
// And each 13 rows it writes
// 8 frames of 26 ms
// 1 frame of 27 ms
//It is done for adjusting the different durations of each frame.
// as they vary between 26.125 ms and 26.075 ms

swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
TimeToSampleBox.sample_count[0] = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
TimeToSampleBox.sample_delta[0] = 0;
int t=0,k=1, l =0;
num_entr = 1;
j = 0;
for (i = 1; i< num_frame; i++) {
    if (j == 8 && l == 0) {
        swap = byterevers(7);
        moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
        TimeToSampleBox.sample_count[num_entr] = swap;
        swap = byterevers(26);
        moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
        TimeToSampleBox.sample_delta[num_entr] =swap;
        num_entr ++;

        swap = byterevers(1);
        moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
        TimeToSampleBox.sample_count[num_entr] = swap;
        swap = byterevers(27);
        moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
        TimeToSampleBox.sample_delta[num_entr] =swap;
        num_entr++;
        j=0;
        dat = i;
        if (k == 6 && t == 0) {
            l = 1;
            t = 1;
            k = 1;
        }
        if (k == 6 && t ==1) {
            l = 1;

```



```

        k = 1;
    }
    k++;
}

if (j == 9 && l == 1) {

    swap = byterevers(8);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.sample_count[num_entr] = swap;
    swap = byterevers(26);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.sample_delta[num_entr] = swap;
    num_entr ++;

    swap = byterevers(1);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.sample_count[num_entr] = swap;
    swap = byterevers(27);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
    TimeToSampleBox.sample_delta[num_entr] = swap;
    num_entr++;
    j=0;
    dat = i;
    l = 0;
}
j++;
}

dat = num_frame - dat;

swap = byterevers(dat);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
TimeToSampleBox.sample_count[num_entr] = swap;
swap = byterevers(26);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
TimeToSampleBox.sample_delta[num_entr] = swap;
num_entr++;
swap = byterevers(num_entr);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SampleTableBox.
TimeToSampleBox.entry_count = swap;

fclose(song);
return num_frame;
}

```

## Track Structure

```

int trackstructure (MovieBox *moov, int numtrack, int clock,
                    int durationTrack, int sizemdat, char name[20]){
    int swap;

    //Sample Table Box
    int sizeSTBL = 0;
    sizeSTBL = samplecontainer(moov, numtrack, sizemdat, name);

    //Data Entry Url Box
    u32 sizeURL = 12;
    swap = byterevers(sizeURL);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
    DataReferenceBox.DataEntryUrlBox.size = swap;
    swap = byterevers('url ');
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
    DataReferenceBox.DataEntryUrlBox.type = swap;
    swap = byterevers(1);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
    DataReferenceBox.DataEntryUrlBox.flags = swap; // =1 Track in same file as movie
atom.

    //Data Reference
    u32 sizeDREF = sizeURL+ 16;
    swap = byterevers(sizeDREF);
    moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
    DataReferenceBox.size = swap;
}

```

```

swap = byterevers('dref');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
DataReferenceBox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
DataReferenceBox.flags = 0;
swap = byterevers(1);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.
DataReferenceBox.entry_count = swap;

//Data information Box//
u32 sizeDINF = sizeDREF + 8;
swap = byterevers(sizeDINF);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.size =
swap;
swap = byterevers('dinf');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.DataInformationBox.type =
swap;

//Sound Header Box //
u32 sizeSMHD = 16;
swap = byterevers(sizeSMHD);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SoundMediaHeaderBox.size =
swap;
swap = byterevers('smhd');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SoundMediaHeaderBox.type =
swap;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SoundMediaHeaderBox.version
= 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SoundMediaHeaderBox.balance
= 0;
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.SoundMediaHeaderBox.reserved
= 0;

//Media Information Box//
u32 sizeMINF = sizeDINF + sizeSMHD + sizeSTBL + 8;
swap = byterevers(sizeMINF);
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.size = swap;
swap = byterevers('minf');
moov->TrackBox[numtrack].MediaBox.MediaInformationBox.type = swap;

//Handler Box//
u32 sizeHDLR = 37;
swap = byterevers(sizeHDLR);
moov->TrackBox[numtrack].MediaBox.HandlerBox.size = swap;
swap = byterevers('hdlr');
moov->TrackBox[numtrack].MediaBox.HandlerBox.type = swap;
moov->TrackBox[numtrack].MediaBox.HandlerBox.version = 0;
moov->TrackBox[numtrack].MediaBox.HandlerBox.pre_defined = 0;
swap = byterevers('soun');
moov->TrackBox[numtrack].MediaBox.HandlerBox.handler_type = swap;
moov->TrackBox[numtrack].MediaBox.HandlerBox.reserved[0] = 0;
moov->TrackBox[numtrack].MediaBox.HandlerBox.reserved[1] = 0;
moov->TrackBox[numtrack].MediaBox.HandlerBox.reserved[2] = 0;
//swap = byterevers('soun');
//moov->TrackBox[numtrack].MediaBox.HandlerBox.data = swap;
moov->TrackBox[numtrack].MediaBox.HandlerBox.data[0] = 's';
moov->TrackBox[numtrack].MediaBox.HandlerBox.data[1] = 'o';
moov->TrackBox[numtrack].MediaBox.HandlerBox.data[2] = 'u';
moov->TrackBox[numtrack].MediaBox.HandlerBox.data[3] = 'n';
moov->TrackBox[numtrack].MediaBox.HandlerBox.data[4] = '\0';

//Media Header Box//
u32 sizeMDHD = 32;
swap = byterevers(sizeMDHD);
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.size = swap;
swap = byterevers('mdhd');
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.type = swap;
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.version = 0;
swap = byterevers(clock);
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.creation_time = swap;
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.modification_time = swap;
swap = byterevers(1000);
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.timescale = swap;
swap = byterevers(durationTrack);

```

```

moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.duration = swap;
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.language = 0xC455;
moov->TrackBox[numtrack].MediaBox.MediaHeaderBox.pre_defined = 0;

//Media Box//
u32 sizeMDIA = sizeMDHD + sizeHDLR + sizeMINF + 8;
swap = byterevers(sizeMDIA);
moov->TrackBox[numtrack].MediaBox.size = swap;
swap = byterevers('mdia');
moov->TrackBox[numtrack].MediaBox.type = swap;

//Track Header//
u32 sizeTKHD = 92;
swap = byterevers (sizeTKHD);
moov->TrackBox[numtrack].TrackHeaderBox.size = swap;
swap = byterevers ('tkhd');
moov->TrackBox[numtrack].TrackHeaderBox.type = swap ;
swap = byterevers (0x00000006);
moov->TrackBox[numtrack].TrackHeaderBox.version = swap;
swap = byterevers (clock);
moov->TrackBox[numtrack].TrackHeaderBox.creation_time = swap;
moov->TrackBox[numtrack].TrackHeaderBox.modification_time = swap;
swap = byterevers (numtrack+1);
moov->TrackBox[numtrack].TrackHeaderBox.track_ID = swap; //From 0x00000001 -
0x7FFFFFFF (dec 2147483647)
moov->TrackBox[numtrack].TrackHeaderBox.reserved = 0;
swap = byterevers (durationTrack);
moov->TrackBox[numtrack].TrackHeaderBox.duration = swap;
moov->TrackBox[numtrack].TrackHeaderBox.reserved2[0] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.reserved2[1] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.layer = 0;
moov->TrackBox[numtrack].TrackHeaderBox.alternate_group = 0;
moov->TrackBox[numtrack].TrackHeaderBox.volume = 0x1;
moov->TrackBox[numtrack].TrackHeaderBox.reserved3 = 0;
swap = byterevers (0x00010000);
moov->TrackBox[numtrack].TrackHeaderBox.matrix[0] = swap;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[1] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[2] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[3] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[4] = swap;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[5] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[6] = 0;
moov->TrackBox[numtrack].TrackHeaderBox.matrix[7] = 0;
swap = byterevers(0x40000000);
moov->TrackBox[numtrack].TrackHeaderBox.matrix[8] = swap;
moov->TrackBox[numtrack].TrackHeaderBox.width = 0; //just for video
moov->TrackBox[numtrack].TrackHeaderBox.height = 0; //just for video

//Track container
u32 sizeTRAK = sizeTKHD + sizeMDIA + 8;
swap = byterevers (sizeTRAK); // Size of one track
moov->TrackBox[numtrack].size = swap;
swap = byterevers ('trak');
moov->TrackBox[numtrack].type = swap;
return sizeTRAK;
}

```

## Preset Container

```

int presetcontainer(MovieBox *moov, int totaltracks, nametrack namet){

    int swap, i,j,flag, vol=0;
    unsigned char numpres=0, prestype=0,defaultPreset;
    char namepres1[14] = "static_track "; // 13
    u32 sizePRST = 0;

    printf("\nPresets:\n");
    printf("Static track volume preset: invariant volume related to each track \n");
    printf("-----\n");
    numpres = 1;

    //Preset Box//
    for (i=0; i<numpres; i++) {
        printf("Preset number %d: %s\n",i+1,namepres1);
    }
}

```

```

strcpy(moov->PresetContainerBox.PresetBox[i].preset_name, namepres1);
sizePRST = 16 + 14 + 4*totaltracks + totaltracks;
swap = byterevers(sizePRST);
moov->PresetContainerBox.PresetBox[i].size = swap;
prestype = 0;

moov->PresetContainerBox.PresetBox[i].num_preset_elements = totaltracks;
swap = byterevers('prst');
moov->PresetContainerBox.PresetBox[i].type = swap;

flag = 0x02; // Display Enable Edit Disable
swap = byterevers(flag);
moov->PresetContainerBox.PresetBox[i].flags = swap;

moov->PresetContainerBox.PresetBox[i].preset_ID = i+1;

moov->PresetContainerBox.PresetBox[i].preset_type = prestype;
moov->PresetContainerBox.PresetBox[i].preset_global_volume = 100;

for (j=0; j<totaltracks; j++) {
    swap = byterevers(j+1);
    moov->PresetContainerBox.PresetBox[i].presElemId[j].preset_element_ID =
swap;
}
//Enter values (two options):
// In loop
/*    for (j=0; j<totaltracks; j++) {
        vol = 70 - 10*(j+1);
        vol = 20*(j+1);
        printf("Enter volume for %s = %d\n",namet[j].title,vol*2);
        scanf("%d",&vol);
        printf("Vol: %d\n",vol);
        moov->PresetContainerBox.PresetBox[i].presVolumElem[j].preset_volume_element = vol; /*0.02
    }
*/
    // Or one by one
    moov->PresetContainerBox.PresetBox[i].presVolumElem[0].preset_volume_element =
20;
    printf("Enter volume for %s = %d\n",namet[j].title,moov->PresetContainerBox.
        PresetBox[i].presVolumElem[0].preset_volume_element*2);
    moov->PresetContainerBox.PresetBox[i].presVolumElem[1].preset_volume_element =
40;
    printf("Enter volume for %s = %d\n",namet[j].title,moov->PresetContainerBox.
        PresetBox[i].presVolumElem[1].preset_volume_element*2);
    moov->PresetContainerBox.PresetBox[i].presVolumElem[2].preset_volume_element =
20;
    printf("Enter volume for %s = %d\n",namet[j].title,moov->PresetContainerBox.
        PresetBox[i].presVolumElem[2].preset_volume_element*2);
    moov->PresetContainerBox.PresetBox[i].presVolumElem[3].preset_volume_element =
50;
    printf("Enter volume for %s = %d\n",namet[j].title,moov->PresetContainerBox.
        PresetBox[i].presVolumElem[3].preset_volume_element*2);
    moov->PresetContainerBox.PresetBox[i].presVolumElem[4].preset_volume_element =
52;
    printf("Enter volume for %s = %d\n",namet[j].title,moov->PresetContainerBox.
        PresetBox[i].presVolumElem[4].preset_volume_element*2);
}

//Preset Container//
u32 sizePRCO = sizePRST + 10;
swap = byterevers(sizePRCO);
moov->PresetContainerBox.size = swap;
swap = byterevers('prco');
moov->PresetContainerBox.type = swap;
defaultPreset = 1;
moov->PresetContainerBox.default_preset_ID = defaultPreset; // Indicates initial
preset activated.
moov->PresetContainerBox.num_preset = numpres;

return sizePRCO;
}

```

## Rules Container

```

int rulescontainer(MovieBox *moov){
    int swap;
    u32 sizeRUSC, elementID, key_elem, sizeRUMX;

    moov->RulesContainer.num_selection_rules = 256; //u16 invert
    // moov->RulesContainer.num_selection_rules = 0;
    moov->RulesContainer.num_mixing_rules = 256; //u16 invert

    //Selection Rules
    sizeRUSC = 19 + 14;
    // sizeRUSC = 0;
    swap = byterevers(sizeRUSC);
    moov->RulesContainer.SelectionRules.size = swap;
    swap = byterevers('rusc');
    moov->RulesContainer.SelectionRules.type = swap;
    moov->RulesContainer.SelectionRules.version = 0;
    moov->RulesContainer.SelectionRules.selection_rule_ID = 256;
    moov->RulesContainer.SelectionRules.selection_rule_type = 2;
    elementID = 4;
    swap = byterevers(elementID);
    moov->RulesContainer.SelectionRules.element_ID = swap;
    strcpy(moov->RulesContainer.SelectionRules.rule_description, "Not mute rule");
    printf("\nRules:\n");
    printf("Rule 1: Not mute for channel %d\n", elementID);

    //Mixing Rule
    sizeRUMX = 23 + 17;
    swap = byterevers(sizeRUMX);
    moov->RulesContainer.MixingRules.size = swap;
    swap = byterevers('rumx');
    moov->RulesContainer.MixingRules.type = swap;
    moov->RulesContainer.MixingRules.version = 0;
    moov->RulesContainer.MixingRules.mixing_rule_ID = 512;
    // moov->RulesContainer.MixingRules.mixing_type = 0; // Equivalence rule
    moov->RulesContainer.MixingRules.mixing_type = 2; // Upper rule
    elementID = 5;
    swap = byterevers(elementID);
    moov->RulesContainer.MixingRules.element_ID = swap;
    key_elem = 3;
    swap = byterevers(key_elem);
    moov->RulesContainer.MixingRules.key_elem_ID = swap;
    strcpy(moov->RulesContainer.MixingRules.mix_description, "Lower rule");
    printf("Rule 2: Lower rule between channel %d and %d\n", elementID, key_elem);
    // strcpy(moov->RulesContainer.MixingRules.mix_description, "Equivalence rule");
    // printf("Rule 2: Equivalence rule\n");

    //Rule container
    u32 sizeRUC0 = 12 + sizeRUSC + sizeRUMX;
    swap = byterevers(sizeRUC0);
    moov->RulesContainer.size = swap;
    swap = byterevers('ruco');
    moov->RulesContainer.type = swap;

    return sizeRUC0;
}

```

## Movie header box

```

void moovheaderbox (MovieBox *moov, int clock, int sizeTRAK, int sizePRC0, int
totaltracks, int durationTrack, int sizeRUC0){
    int swap;

    //MovieHeader
    u32 sizeMVHD = 108;
    swap = byterevers (sizeMVHD);
    moov->MovieHeaderBox.size = swap;
    swap = byterevers ('mvhd');
    moov->MovieHeaderBox.type = swap;
    moov->MovieHeaderBox.version = 0;
    swap = byterevers (clock);
    moov->MovieHeaderBox.creation_time = swap;
    moov->MovieHeaderBox.modification_time = swap;
    swap = byterevers (1000);
    moov->MovieHeaderBox.timescale = swap;
    swap = byterevers (durationTrack);
}

```

```

moov->MovieHeaderBox.duration = swap;
swap = byterevers (0x00010000);
moov->MovieHeaderBox.rate = swap;
swap = byterevers (1);
moov->MovieHeaderBox.volume = 1;
moov->MovieHeaderBox.reserved=0;
moov->MovieHeaderBox.reserved2[0] = 0;
moov->MovieHeaderBox.reserved2[1] = 0;
swap = byterevers (0x00010000);
moov->MovieHeaderBox.matrix[0] = swap;
moov->MovieHeaderBox.matrix[1] = 0;
moov->MovieHeaderBox.matrix[2] = 0;
moov->MovieHeaderBox.matrix[3] = 0;
moov->MovieHeaderBox.matrix[4] = swap;
moov->MovieHeaderBox.matrix[5] = 0;
moov->MovieHeaderBox.matrix[6] = 0;
moov->MovieHeaderBox.matrix[7] = 0;
swap = byterevers (0x40000000);
moov->MovieHeaderBox.matrix[8] = 0x40000000;
moov->MovieHeaderBox.pre_defined[0] = 0;
moov->MovieHeaderBox.pre_defined[1] = 0;
moov->MovieHeaderBox.pre_defined[2] = 0;
moov->MovieHeaderBox.pre_defined[3] = 0;
moov->MovieHeaderBox.pre_defined[4] = 0;
moov->MovieHeaderBox.pre_defined[5] = 0;
swap = byterevers (totaltracks + 1);
moov->MovieHeaderBox.next_track_ID = swap;

//MovieBox
u32 sizeMOOV = sizeMVHD + sizeTRAK + sizePRCO + sizeRUCO + 8;
swap = byterevers (sizeMOOV); //Size movie: Taking into account number tracks
moov->size = swap;
swap = byterevers ('moov');
moov->type = swap;
}

```

## IM AF encoder.h

```
//
// IM_AF Encoder.h
// IM_AM Encoder
//
// Created by Eugenio Oñate Hospital on 14/06/12.
// Copyright (c) 2012 SAE. All rights reserved.
//

#ifndef IM_AM_Encoder_IM_AF_Encoder_h
#define IM_AM_Encoder_IM_AF_Encoder_h

/* for va_list typedef */
#include <stdarg.h>
/* for FILE typedef, */
#include <stdio.h>

#define maxtracks 8
#define maxpreset 10
#define maxrules 10

typedef long long u64;
typedef unsigned int u32;
typedef unsigned short u16;

typedef struct nametrack { // Stores the different titles of the tracks
    char title[20];
}nametrack[maxtracks];

typedef struct FileTypeBox
{
    u32 size;
    u32 type;           // ftyp
    u32 major_brand;    // brand identifier
    u32 minor_version; // informative integer for the mirror version
    u32 compatible_brands[2]; //list of brands
}FileTypeBox;

typedef struct MoiveBox //extends Box('moov')
{
    u32 size;
    u32 type;           // moov

    struct MovieHeaderBox
    {
        u32 size;
        u32 type; // mvhd
        u32 version; // version + flag
        u32 creation_time;
        u32 modification_time;
        u32 timescale; // specifies the time-scale
        u32 duration;
        u32 rate; // typically 1.0
        u16 volume; // typically full volume
        u16 reserved; // =0
        u32 reserved2[2]; // =0
        u32 matrix[9]; // information matrix for video (u,v,w)
        u32 pre_defined[6]; // =0
        u32 next_track_ID; //non zero value for the next track ID
    }MovieHeaderBox;

    struct TrackBox
    {
        u32 size;
        u32 type;
        struct TrackHeaderBox
        {
            u32 size;
            u32 type;
            u32 version; // version + flag
            u32 creation_time;
            u32 modification_time;
            u32 track_ID;
            u32 reserved; // =0
        }
    }
}
```

```

    u32 duration;
    u32 reserved2[2]; // =0
    u16 layer; // =0 // for video
    u16 alternate_group; // =0
    u16 volume; // full volume is 1 = 0x0100
    u16 reserved3; // =0
    u32 matrix[9]; // for video
    u32 width; // video
    u32 height; // video
}TrackHeaderBox;

struct MediaBox // extends Box('mdia')
{
    u32 size;
    u32 type;
    struct MediaHeaderBox // extends FullBox('mdhd', version,0)
    {
        u32 size;
        u32 type;
        u32 version; // version + flag
        u32 creation_time;
        u32 modification_time;
        u32 timescale;
        u32 duration;
        u16 language; // [pad,5x3] = 16 bits and pad = 0
        u16 pre_defined; // =0
    }MediaHeaderBox;
    struct HandlerBox
    {
        u32 size;
        u32 type;
        u32 version; // version = 0 + flag
        u32 pre_defined; // =0
        u32 handler_type; // = 'soun' for audio track, text or hint
        u32 reserved[3]; // =0
        unsigned char data[5]; // Does not work! only 4 bytes
    }HandlerBox;
    struct MediaInformationBox //extends Box('minf')
    {
        u32 size;
        u32 type;
        // smhd in sound track only!!
        struct SoundMediaHeaderBox
        {
            u32 size;
            u32 type;
            u32 version;
            u16 balance; // =0 place mono tracks in stereo. 0 is center
            u16 reserved; // =0
        }SoundMediaHeaderBox;

        struct DataInformationBox //extends Box('dinf')
        {
            u32 size;
            u32 type;
            struct DataReferenceBox
            {
                u32 size;
                u32 type;
                u32 flags;
                u32 entry_count; // counts the actual entries.
                struct DataEntryUrlBox //extends FullBox('url', version=0,
flags)
                {
                    u32 size;
                    u32 type;
                    u32 flags;
                }DataEntryUrlBox;
            }DataReferenceBox;
        }DataInformationBox;
        struct SampleTableBox // extends Box('stbl')
        {
            u32 size;

```



```

u32 type;
struct TimeToSampleBox{
    u32 size;
    u32 type;
    u32 version;
    u32 entry_count;
    u32 sample_count[3000];
    u32 sample_delta[3000];
}TimeToSampleBox;
struct SampleDescriptionBox // stsd
{
    u32 size;
    u32 type;
    u32 version;
    u32 entry_count; // = 1 number of entries
    // unsigned char esds[88];
    struct AudioSampleEntry{
        u32 size;
        u32 type; //mp4a
        char reserved[6];
        u16 data_reference_index; // = 1
        u32 reserved2[2];
        u16 channelcount; // = 2
        u16 samplesize; // = 16
        u32 reserved3;
        u32 samplerate; // 44100 << 16
    // unsigned char esds[81];
    struct ESbox{
        u32 size;
        u32 type;
        u32 version;
        struct ES_Descriptor{
            unsigned char tag;
            unsigned char length;
            u16 ES_ID;
            unsigned char mix;
            struct DecoderConfigDescriptor{
                unsigned char tag;
                unsigned char length;
                unsigned char objectProfileInd;
                u32 mix;
                u32 maxBitRate;
                u32 avgBitrate;
                /* struct DecoderSpecificInfo{
                    unsigned char tag;
                    unsigned length;
                    // unsigned char decSpecificInfosize;
                    unsigned char decSpecificInfoData[2];
                }DecoderSpecificInfo;
            }DecoderConfigDescriptor;
            struct SLConfigDescriptor{
                unsigned char tag;
                unsigned char length;
                unsigned char predefined;
            }SLConfigDescriptor;
        }ES_Descriptor;
    }ESbox;
    }AudioSampleEntry;
}SampleDescriptionBox;
struct SampleSizeBox{
    u32 size;
    u32 type;
    u32 version;
    u32 sample_size; // =0
    u32 sample_count;
    u32 entry_size[9000];
}SampleSizeBox;
struct SampleToChunk{
    u32 size;
    u32 type;
    u32 version;
    u32 entry_count;
    u32 first_chunk;
    u32 samples_per_chunk;

```

```

        u32 sample_description_index;
    }SampleToChunk;
    struct ChunkOffsetBox{
        u32 size;
        u32 type;
        u32 version;
        u32 entry_count;
        u32 chunk_offset[maxtracks];
    }ChunkOffsetBox;
    }SampleTableBox;
    }MediaInformationBox;
}MediaBox;
}TrackBox[maxtracks]; // max 10 tracks
struct PresetContainerBox // extends Box('prco')
{
    u32 size;
    u32 type;
    unsigned char num_preset;
    unsigned char default_preset_ID;
    struct PresetBox //extends FullBox('prst',version=0,flags)
    {
        u32 size;
        u32 type;
        u32 flags;
        unsigned char preset_ID;
        unsigned char num_preset_elements;
        struct presElemId{
            u32 preset_element_ID;
        }presElemId[maxtracks];
        unsigned char preset_type;
        unsigned char preset_global_volume;
        //IF preset_type == 1
        struct presVolumElem{
            unsigned char preset_volume_element;
        }presVolumElem[maxtracks];
        char preset_name[14];
    }PresetBox[maxpreset];
}PresetContainerBox;

struct RulesContainer{
    u32 size;
    u32 type;
    u16 num_selection_rules;
    u16 num_mixing_rules;
    struct SelectionRules{
        u32 size;
        u32 type;
        u32 version;
        u16 selection_rule_ID;
        unsigned char selection_rule_type;
        u32 element_ID;
        char rule_description[14];
    }SelectionRules;
    struct MixingRules{
        u32 size;
        u32 type;
        u32 version;
        u16 mixing_rule_ID;
        unsigned char mixing_type;
        u32 element_ID;
        u32 key_elem_ID;
        char mix_description[17];
    }MixingRules;
    }RulesContainer;
}MovieBox;
typedef struct MediaDataBox // extends Box('mdat')
{
    u32 size;
    u32 type;
    unsigned char data;
}MediaDataBox;
#endif

```