SECONDA UNIVERSITA' DEGLI STUDI DI NAPOLI

Dipartimento di Ingegneria Industriale e dell'Informazione

LAUREA MAGISTRALE

IN

INGEGNERIA ELETTRONICA

MPEG-A Interactive Music Application Format (IM AF) Encoder

- Features Development -

Relatore:
Ch.mo Prof.
Gianmarco Romano

Candidato:
Costantino Taglialatela
Matr. A17 000 028

Anno Accademico 2012-2013

Alla mia famiglia,
per tutto il sostegno ricevuto.

Al Prof. Gianmarco Romano,
per la sua fiducia e per i preziosi consigli
ricevuti durante tutto il percoso di studio.

Ad Antonella,
per la pazienza e l'amore,
per me essenziali in tutti questi anni.

ACKNOWLEDGEMENT

The author acknowledges that this work has been done during his visit at the Centre for Digital Music (C4DM) - School of Electronic Engineering and Computer Science, Queen Mary University of London (EECS-QMUL).

His deepest gratitude goes to Prof. Panos Kudumakis for his supervision and to Dr. Mark Plumbley for the invaluable opportunity of the visit in C4DM.

The author would also like to express his thankfulness to Chris Cannam and Luis Figueira for their assistance and the precious suggestions.

The first version of the IM AF Encoder has been proposed by Eugenio Oñate (EECS-QMUL) in August 2012. This work is based on his proof of concept, in collaboration with Jesús Corral García (E.T.S.I. Telecomunicación, Dpt. Ingeniería de Comunicaciones, Universidad de Málaga).

INDEX

INTRODUCTION

The fact that the MP3 file format reigns supreme in the music market is not news. Despite having its market position constantly "attacked" by new formats such as AAC, Ogg Vorbis and so on, nothing has changed so far [16].

In fact, users continue to play the part of "passive listeners", in disagreement with the interactivity trend brought by recent devices, such as smartphones and tablets. Keeping in mind the continuous development of apps like *Glee Karaoke* and *I am T-Pain* [23], and the success of some videogames like *Guitar Hero* [24] and *Rock Band* [25], we can understand that there has been a deeply change in the way consumers perceive music nowadays.

So far, listeners didn't have the opportunity to modify the songs and adapt them to their taste unless they had in their disposition a recording studio with expensive gear and the multiple tracks which form the songs.

In this context, new interactive music services have emerged with the aim of enhancing the listeners' experience. However, each service uses its own proprietary file format and this has made it difficult to boost the global interactive music market. A standardized file format is inevitably required to provide the interoperability between various interactive music players and interactive music songs and albums [15].

This issue is addressed in a recently published standard by ISO/IEC Moving Picture Experts Group (MPEG) known as *MPEG-A: Interactive Music Application Format* (*IM AF*) [1]. It is a *multitrack* based audio file format, that means that it can contain separate sound elements for a song (like individual tracks for drums, guitar, vocals, etc.). It allows users to modify the mixing style of the song, by changing the volume of each music instrument separately.

Other multitrack and interactive formats have already been available, e.g., MOGG [19], IEEE 1599 [17] and iXMF [11]. The novelty of IM AF is on additional media data that enrich the user's interaction space, like *timed text* synchronized with audio tracks, which can represent the lyrics or the chords of a song, and *images* related to the song, album and artist. Another important feature is the introduction of *presets* (e.g., a karaoke or rhythmic version of the song offered by the music producer) and *rules* constraining the user mixing result to preserve the artistic creation of the composer.

The IM AF specifications come together with a reference software (a non-optimized) *decoder* and *conformance files*. In this way MPEG enables companies to build IM AF compliant encoders and offer associated services and products to market. However no one has publicly released an implementation for an encoder yet, although commercial services exist [14].

This work comes to fill in this gap, proposing an implementation of the very first full-fledged codec compliant to the IM AF file format.

Availability of software that allows creating and playing IM AF files could revitalize the music market. New on-line music forums and social networks could be handy from this point of view: personal mixes of songs could be exported and easily shared between users, having lighter files that contain only information about the mixing parameters while the audio tracks can be made

available through various on-line music services. Each audio track could even be replaced by users' personal recordings, encouraging people to develop singing and music instruments playing skills through active learning.

In particular, the rest of this thesis is structured as follows: an overview about previously released multitrack/interactive music files is presented in ***Chapter 1***, together with a comparison of these formats with IM AF. The IM AF standard and the file structure are described in ***Chapter 2***, while the implementation of the encoder is described in ***Chapter 3***. In ***Chapter 4***, some example IM AF files are presented as the result of the work, to demonstrate the compliancy of the encoder to the standard. Furthermore, ***Chapter 5*** describes the integration of the developed encoder in *Sonic Visualiser* [9] (an application for viewing and analysing the contents of music audio files) and shows some use-cases of plugins when the IM AF support is provided. Some final remarks are presented in ***Chapter 6***.

A first version of an *IM AF Encoder* was released by the *Centre for Digital Music* (C4DM) of the Queen Mary University of London [26], as a proof of concept, with basic capabilities and allowing the inclusion of only audio tracks in the resulting IM AF file.

This work is a continuation of the previous one about the IM AF Encoder and dealt with the following tasks:

- *Add pictures to be displayed during the playing of a song;*
- *Add timed-text as lyrics (supported by Jesus Corral Garcia);*
- *Inclusion of groups, presets and rules as described by the standard;*
- *Test of encoder's full compliancy to the standard;*
- *Provide IM AF support to Sonic Visualiser (export only).*

CHAPTER I

BACKGROUND RESEARCH

Introduction

Having a look at previous attempts to create an interactive music file is useful to understand the context in which the IM AF file format is born.

In this section, some already available interactive and multitrack audio file formats for the digital music market are briefly described. Some of them were released by the same creators of the Interactive Music Application Format.

Many of the services and file formats that will be mentioned in the section are only known to a limited circle of users. Also some of them are not survived to their test period, due to various policies of the companies and the market.

IM AF is a pretty young standard and it has potential to have a better fortune.

1.1 Interactive music services and file formats

The consumer's continuous search for different way to enjoy their favourite music incites developers to create new interesting services. No more than 5 years ago many different interactive music services came into the world, each one using proprietary file formats with dissimilar features and no compatibility between them.

Two companies in particular, iKlax Media [14] and Audizen [27], gave life to two music services with the aim of revitalizing the music market. In particular, Audizen sponsored its format with the ambitious commercial title of "*Music 2.0*", promising a completely and dramatically changed approach to the audio multimedia experience overall. After a test period for their own services, iKlax and Audizen decided to join forces for the definition of a new standard format, which is the IM AF.

1.1.1 Perform A Track

Perform A Track is the interactive music player based on the *iKlax multi-track technology* [14]. It is possible to buy and download songs from the website and then use them with the player to shut down one or more instruments, listening to the result live. Users can change each stem track volume and then save their mix for a future playback. Mix savings are unlimited.

iKlax is a French research group that in collaboration with LaBRI (the *Laboratoire Bordelais de Recherche en Informatique* from France) developed the file format with the extension '*.iklax*'; Perform A Track is the only one software that is able to play such kind of file. It is available for iPhone/iPad and PC and it is a sort of evolution of the first iKlax player for personal computer.

The online songs database is not wide and the maturity level of the software is low yet. Moreover, despite iKlax Media is one of the major contributor to the creation of the IM AF (they introduced the concepts of *presets*, *rules* and *groups*, and the *Rule Analyser* library used in the reference software), Perform A Track is not yet compliant with the standard and still uses iKlax format; actually, the possibility of having lyrics and chords as timed text data in a song has been recently announced and yet to be introduced. These could be the reasons why the service is still not so popular.

Figure 1.1 - "Perform A Track" music player interface for PC and iPhone/iPad.

1.1.2 Music 2.0

The Audizen company and the Korean governmental research institute ETRI (*Electronics and Telecommunications Research Institute*) started the "*Music 2.0*" service in 2008, based on the *MT9* file format [28].

The distinctive feature of MT9 format is that to be a multitrack audio files with a limited number of 6 tracks (each channel is dedicated to voice, chorus, piano, guitar, bass and drum); the idea was to allow the file format to be easily played on devices with limited processing capabilities, such as mobile phones. In fact,

Samsung Electronics and LG Electronics were both interested in equipping their mobile phones with an MT9 player, but this project was later abandoned, due probably to the choice of the Motion Picture Experts Group (MPEG), the international organization of the digital music and video industry, to make of the MT9 format one of the basis for a new digital music standard [29].

iKlax and ETRI technologies had been used as references for the IM AF standard, mixing the best features of the two file formats (*presets*, *rules*, *groups* and the *Rule Analyser* library from iKlax, *timed text* and *pictures* from MT9). After the first release of the IM AF standard in the late 2010, Audizen closed his music service, probably with the aim of creating a new one based on the new file format.

However, nothing is happened so far: Audizen seems to be silent and iKlax still uses its own proprietary file format, even if IM AF is now a fully-fledged standard.

Figure 1.2 - "Music 2.0" interactive music player by Audizen.

1.1.3 MOGG

The *MOGG* format is a multitrack variant of the *Ogg Vorbis* [19], an open-source container file format that can multiplex a number of independent streams for audio, video, text and metadata.

A MOGG file is essentially a container file with multiple Ogg files in it. It's not a very popular file format, it comes from the open-source audio community and it became more known after that many songs ripped from the *Rock Band* videogame had been available on internet in this format.

The file format can be played only from one player called *Audacity* [30], an open-source digital audio program that can playback and record multiple tracks.

1.1.4 Song Galaxy

Song Galaxy is a web service specialised in the selling of backing tracks for singers and musicians [31]. In August 2008, they introduced the *Multi Tracker* software, an easy-to-use program which uses multi track files in up to 16 individual tracks, with each instrument or instrument group on a separate track.

The file format used by this software is called *MTF* (*Multi Track File*) and is very similar to the MOGG format (every individual stem is encoded in Ogg format).

The main features of Multi Tracker are the possibility to transpose songs up to +/- 12 semi-tones, save settings in a project and export the mix as a Wave or MP3 file, allowing users to sing or play along the song.

It is probably the only one among the mentioned services that is still active and releases recent songs in multitrack format.

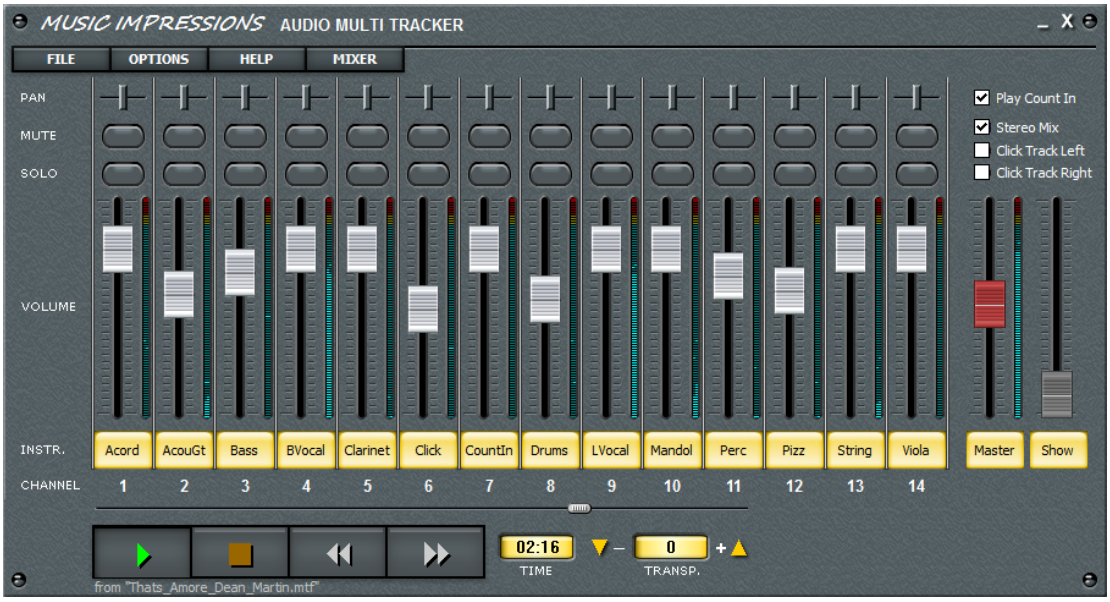

Figure 1.3 - "Multi Tracker" player by Song Galaxy.

1.1.5 MXP4

Another pioneer of interactive music formats is the *MXP4*, developed by the same name MXP4 company, based in Paris and founded in 2008 [32]. The MXP4 format allows to package multimedia content in a single file: audio and video streaming, biographical information about the artist and concert dates. The MXP4 applications also offer interactive features that allow users to play, remix and sing on different tracks, sharing the on social networks.

Despite being an interesting project, in the end of 2010, the company changed its activity and decided to focus on the development of musical social games on Facebook, like *Bopler* [33] (very similar to *Guitar Hero* and *Rock Band* videogames).

1.1.6 iXMF

Interactive XMF is a file format that contains audio content for adaptive audio, introduced by the Interactive Audio Special Interest Group (IASIG) and based on the open-standard XMF (*eXtensible Music Format*) [11].

XMF is a low-overhead, meta file format that includes collections of data resources in one or more formats into a single file; it was developed by the MIDI Manufacturers Association and published in October of 2001.

iXMF is intended as an open cross-platform standard to be used by audio artists, in order to bundle audio content files with general information and audio implementation instructions. It is basically a wrapper format for MIDI files, Downloadable Sounds (DLS), and WAVE waveform data.

The file has an internal structure of "folders" and "files", like a computer file system. It is based on *cue sheets*, that are lists of events corresponding to predefined actions; an event can be triggered at a particular moment in time. An iXMF file may be described in terms of the following abstractions, all represented as data structures stored inside the file:

- An iXMF file is a collection of any number of named Cues;
- A Cue is a collection of Media Chunks, plus some Scripts (rules governing how they are played) ;
- A Media Chunk is a contiguous region in a playable Media File.

Initially, it was meant to be highly beneficial for the game industry: the file format had the backing of Sony Computer Entertainment and it was nearly adopted as one of the supported formats for the PlayStation3 games, but in the end the project was abandoned. Anyway, it may be used in any interactive audio application.

This file format put artistic control into the hands of the artists, keeping programmers from having to make artistic decisions, eliminate rework for porting to new platforms, and reduce production time, cost, and stress.

1.1.7 MOD

A sort of forefather of the iXMF is the *MOD* file format [34]. It is a computer file format used primarily to represent music, and was the first module file format, used on Amiga system in the late 1980s. This format originated a family of music file formats called "*Module files*" that became very popular in the *Demoscene* (a computer-art subculture specialised in audio-visual presentations).

A MOD file contains a set of instruments in the form of samples, a number of patterns indicating how and when the samples are to be played, and a list of what patterns to play in what order. A pattern is typically represented in a sequencer user interface as a table with one column per channel

1.1.8 IEEE 1599

IEEE 1599 encodes music with *XML* symbols aimed at a comprehensive description of music information [17]. This format has been mainly developed at LIM *(Laboratorio di Informatica Musicale*, Università degli Studi di Milano) and since September 2008, the format is an international standard.

It offers two original characteristics compared to existing music standards:

- the encoding is in the form of symbols that can be read both by machines and humans;
- provides additional information surrounding a piece of music.

All the aspects of music (such as audio and sound, graphical representation, historical data and scores) are synchronized within a multi-layered environment; layers can be accessed both individually and as parts of a whole.

The format consists of six layers that communicate with each other, but there can be multiple instances of the same layer type. Figure 1.4 illustrates the interaction between layers, denoted to as:

- **General**: contains metadata information;
- **Logic**: describes the score symbols;
- **Structural**: defines the interactions between musical objects;
- **Notational**: graphical representation of the score;
- **Performance**: computer-based descriptions of a musical performance;
- **Audio**: digital audio recording.

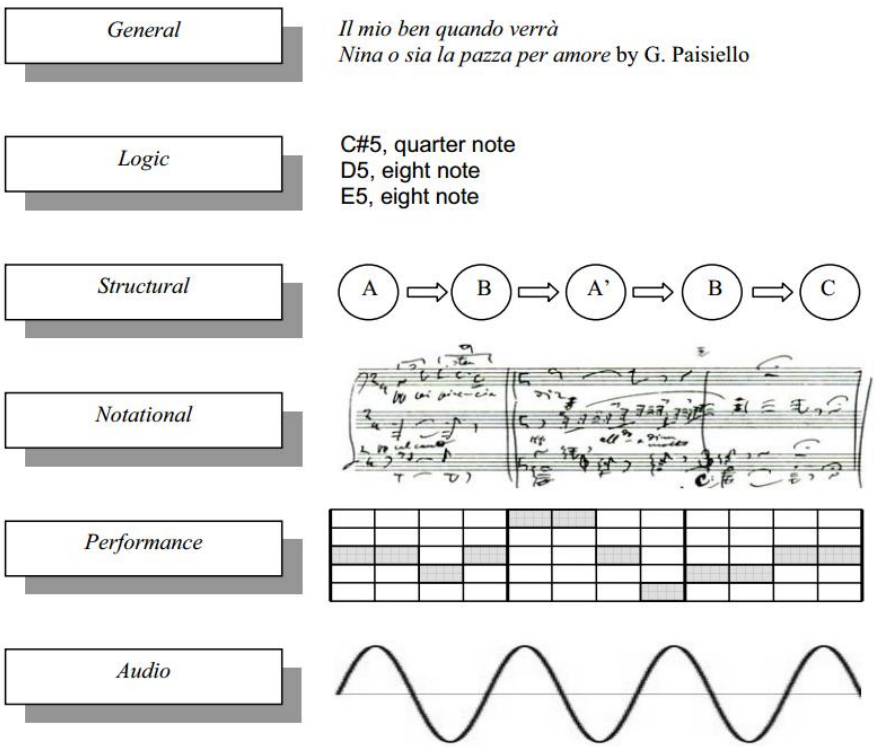

Figure 1.4 - Multi-layered structure of the IEEE 1599 format.

1.2 Connections with IM AF

Each one of the above mentioned formats stand out for a different feature. Despite being developed as a means to organize and describe synchronized streams of information for different applications, a correlation between some of their features can still be found, in order to show the connections with the IM AF file format.

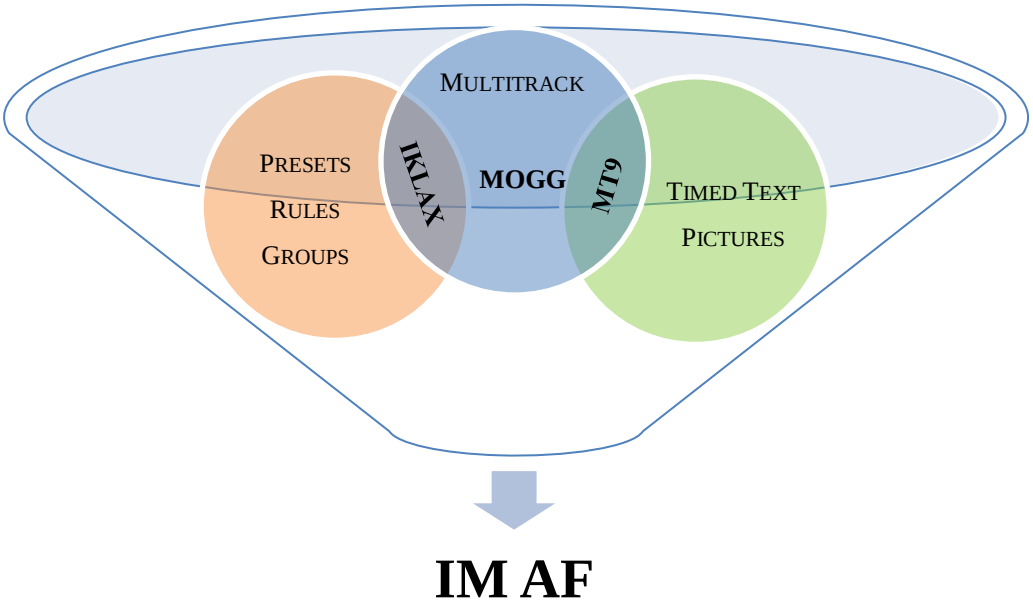

Figure 1.5 - Relationships between features of different file formats and IM AF

Figure 1.5 shows the relationship between Mogg, MT9 (Audizen) and iKlax file formats with IM AF. The other interactive file formats have peculiarities that make them different from the IM AF, and for this reason a comparison doesn't easily stand out. An interesting feature that can be supported by IM AF in a possible future improvement of the standard could be the inclusion of scores, like in the IEEE 1599 format, instead of the simple transcription of chords to display on the timed text track. Option to display videos instead of pictures could also be fascinating, as none of mentioned formats does it.

CHAPTER II

THE MPEG-A INTERACTIVE MUSIC APPLICATION FORMAT (IM AF) STANDARD

Introduction

In this section the features of the IM AF file format are explained in detail, in order to provide a complete understanding of the organization and the playback of an IM AF file.

The *ISO/IEC Moving Picture Experts Group* (MPEG) issued the new standard in 2010 [1], following an amendment published in the late 2012 including extra features [2-3]. The standard's specifications come together with a reference player and few conformance files, enabling developers to create their own software based on the new file format.

The standard embraces different types of media data, especially multiple audio tracks with interactivity data. The structure of the IM AF file is built around the *MPEG-4 ISO Based Media File Format* standard [4]; some improvements had been introduced to enable interactive control over the media data, like *presets*, *rules* and *groups*. Furthermore, additional media like *pictures* and *timed text* are included to enrich the user's interaction space.

2.1 Interactive music services and file formats

There is no better way to understand the possibilities offered by IM AF than exploring the interactivity of the reference software. Using the interactive music player provided with the standard's specifications, users can play a song and handle the tracks that compose it, mainly by editing the volume values.

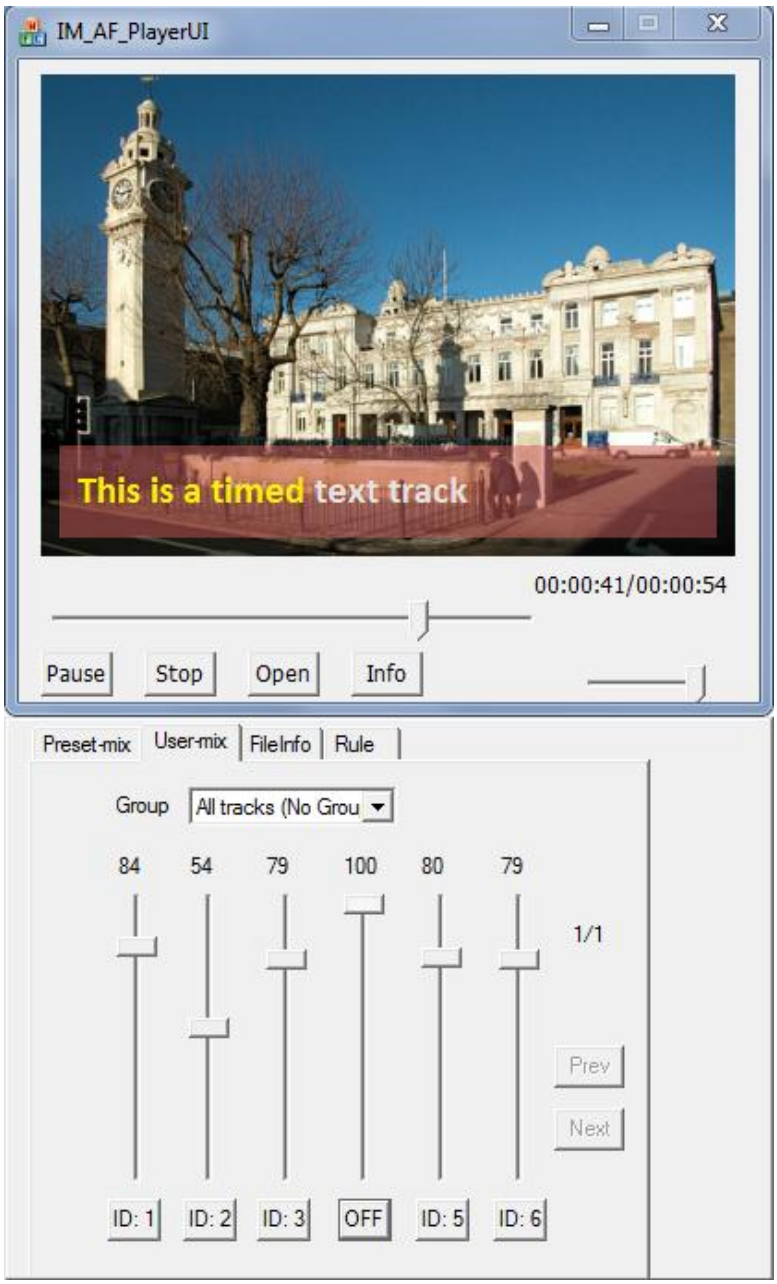

Figure 2.1 - IM AF player user interface.

There are two different modes to interact with the song: the *Preset-Mix mode* and the *User-Mix mode*.

In the Preset-Mix mode, users select one among the *presets* stored in the IM AF file. A *preset* contains predefined information related to the volume of each track in a song. Therefore, giving to a track "zero" volume value, excludes that track from being played in the preset. Some examples for presets could be:

- **Default preset**: the original mix by the music producer;
- **Karaoke preset**: all the tracks except the vocal track are played;
- **Acoustic preset**: only guitar and vocal tracks are played.

In the User-Mix mode, users can control the volume of each track by means of sliders in the player's interface or change their status (*enabled* or *disabled*). It is possible to operate on single tracks or *groups* of them, in order to create "live" custom mixes. Each user's interaction is analysed in order to check the compatibility with the *rules* defined in the file. A *rule* is a constraint decided by the producer and/or the artist to preserve their artistic creation (i.e., for not completely muting the guitar solo in the song).

Then, these two associations can be now assumed:

Preset-Mix mode -> Presets

User-Mix mode -> Groups and Rules

An algorithm called *Rule Analyser* is used to determine compatibility between user's interaction and interactivity rules. Figure 2.2 shows the block diagram for both the Preset-mix mode and the User-mix mode. The preset selection, the audio track/group selection and the volume change are the possible user's interaction.

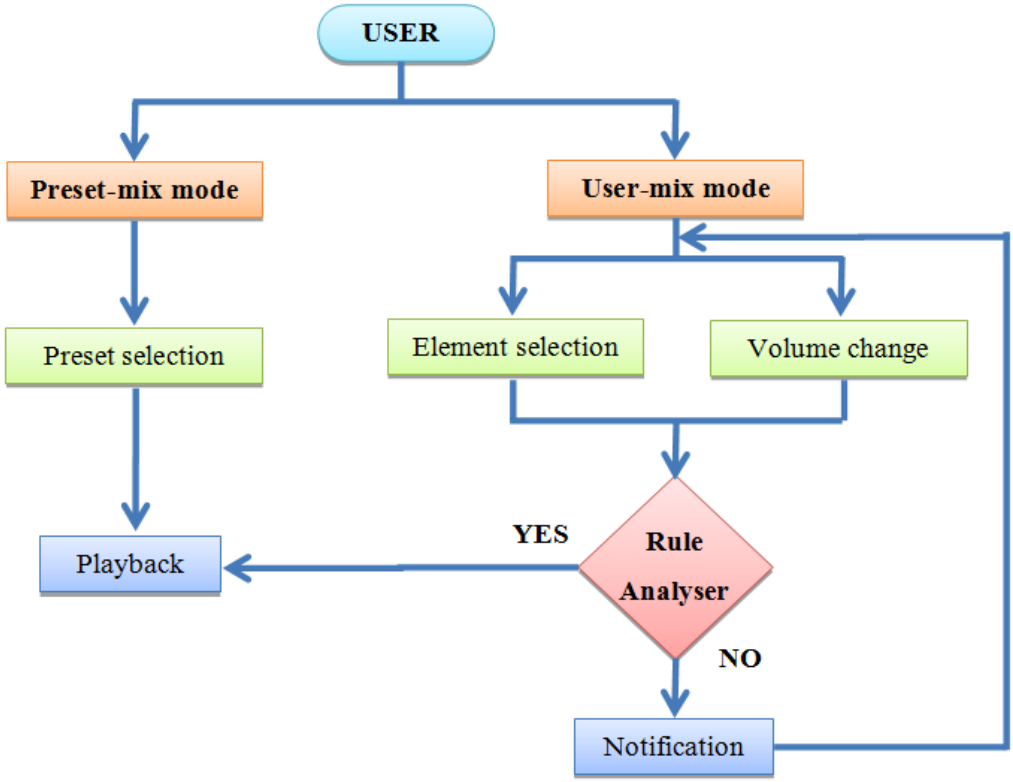

Figure 2.2 - Illustration of the IM AF usage modes.

The rule analyser also interacts with the mixer in order to run user actions when they are conformed to interactivity rules. Actually, the rule analyser consists of a *rule checker* and a *rule solver*.

The rule checker executes rule checking, and rule solver tries to find a solution for the mix using the rule analyser library. If the compatibility is verified, then the actions corresponding to the change of status and volume of the tracks are performed by the Mixer.

In the other case, the rule analyser chooses the most similar valid mix according to the user actions or it simply notifies an error if the requested change is considered impossible to perform.

2.2 File structure

2.2.1 ISO Based Media File Format

The framework of the IM AF file is built around the *MPEG-4 ISO Based Media File Format* (ISO-BMFF) standard [4]; IM AF has introduced some improvements for interactivity control. ISO-BMFF contains the timing, the structure and the media information for timed sequences of media data, such as audio, video and text. A sequence of media data is called *presentation*. The file structure is object-oriented: files conforming to the ISO base media file format are formed as a series of objects, called *boxes*. All data is contained in boxes and there is no other data within the file [35].

An overall structure of an ISO based media file is provided in Table 2.1. Indentation is used to show the hierarchy of the boxes. For example, a *Group Box* '*grup*' is contained in the *Group Container Box* '*grco*', which is included in the *Movie Box* '*moov*'. Not all the boxes in the table must be included in the file. The mandatory boxes are indicated with an asterisk.

There are two different types of boxes: general boxes (simply called *Boxes*), containing data and other boxes, and *FullBoxes* that contain only data. Every box starts with a header defining the *type* and *size* of the box. The *type* is the four character identifier of the each box as shown in the Table 2.1 (i.e., *ftyp*, *moov*, *mdat*); hence it's a 4 bytes value (32 bits, unsigned integer). The *size* is another 32 bits unsigned integer value that indicates the overall size (in bytes) of the box, including data and other possible included boxes.

FullBoxes include in the header two more values: *version* and *flag*. The *version* is an 8 bits integer value that indicates the version of the box; the *flag* is a 24 bits integer value whose use depends on the considered box.

*	**ftyp**						file type and compatibility
*	**moov**						container for all the metadata
		mvhd					movie header, overall declarations
		trak					container for an individual track or stream
*			**trhd**				track header, overall information about the track
			tref				track reference container
			edts				edit list container
				elst			an edit list
*			**mdia**				container for the media information in a track
*				**mdhd**			media header, overall information about the media
*				**hdlr**			handler, declares the media (handler) type "soun" for audio data "text" for timed text data "hint" for protocol hint track
*				**minf**			media information container
					smhd		sound media header, overall information (sound track only)
					hmhd		hint media header, overall information (hint track only)
					nmhd		Null media header, overall information (some tracks only)
*					**dinf**		data information box, container
*						**dref**	data reference box, declares source(s) of media data in track
*					**stbl**		sample table box, container for the time/space map
*						**stsd**	sample descriptions (codec types, initialization etc.)
*						**stts**	(decoding) time-to-sample
*						**stsc**	sample-to-chunk, partial data-offset information
						stsz	sample sizes (framing)
						stz2	compact sample sizes (framing)
*						**stco**	chunk offset, partial data-offset information
						co64	64-bit chunk offset
		grco					container for the groups
			grup				group box, describes the structure (hierarchy)
*		**prco**					container for the presets
*			**prst**				preset box, container for the preset information
		ruco					container for rules
			rusc				selection rule box, container for a selection rule
			rumx				mixing rule box, container for a mixing rule
	mdat						media data container
	free						free space
	skip						free space
	meta						Metadata
*		**hdlr**					handler, declares the metadata (handler) type
		dinf					data information box, container
			dref				data reference box, declares source(s) of metadata items
		iloc					item location
		iinf					item information
		xml					XML container
		bxml					binary XML container
		pitm					primary item reference

Table 2.1- Structure of the boxes in IM AF and their description. The mandatory boxes are marked with (*).

2.2.2 Structure for a single type file

A single type IM AF file contains single movie presentation with associated data. A multiple file type structure can be used to support album functionality (i.e., all the songs from a full album can be stored in one file), but this feature transcend the purpose of this work and therefore it will be discussed at a later stage as a possible future implementation. The *Movie Box* '*moov*', describes the scene presentation. This may include one or more *Track Boxes* '*trak*'. Each '*trak*' box contains the description for one type of media (audio, text or image), while the real media content is stored in the *Media Data Container Box* '*mdat*'. This is illustrated in Figures 2.3.

Alternatively, the '*trak*' box may define a *URL* (*Uniform Resource Locator*) addressing where the media data are stored (i.e., on an internet web server). Thus, IM AF files can be very light in terms of storage requirements mainly consisted of *metadata* (i.e., groups, presets and rules are stored in the file; audio tracks, text and pictures can be streamed by an internet service).

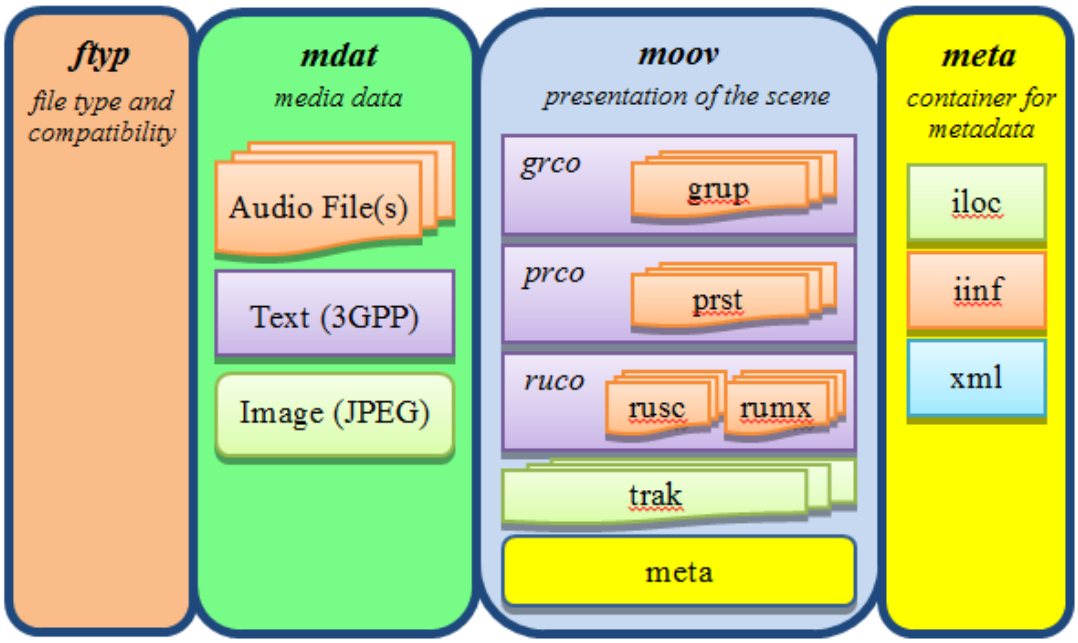

Figure 2.3 - Detailed IM AF file format structure.

2.2.3 Supported standards

The IM AF standard supports compression of the audio tracks in various formats including *PCM*, *MP3*, *AAC*, and *SAOC* [8]. Each track in IM AF format is stored separately from the other tracks, without cross compression on sound signal (specific multichannel compression). This process allows a better quality for sound.

It also supports the *JPEG* file format for still pictures [5], the 3GPP timed text for lyrics [6] and *MPEG-7 Multimedia Description Scheme* for metadata [7]. Table 2.2 lists all the supported components of IM AF with the specification reference of the respective component.

TYPE	COMPONENT NAME	ABBREVIATION	SPECIFICATION
File Format	ISO Base Media File Format	ISO-BMFF	ISO/IEC 14496-12:2008
Audio	MPEG-1 Audio Layer III	MP3	ISO/IEC 11172-3:1993
	MPEG-4 Audio AAC profile	AAC	ISO/IEC 14496-3:2005
	MPEG-D SAOC Baseline profile	SAOC	ISO/IEC 23003-2:2010
	PCM	PCM	-
Text	JPEG Image	JPEG	ISO/IEC 10918-1:1994
Image	3GPP Timed Text	3GPP TT	ETS 3GPP TS 26.245-2004
Metadata	MPEG-7 Multimedia Description Scheme	MDS	ISO/IEC 15938-5:2003

Table 2.2 - Supported components in IM AF.

The IM AF player provides the parsing function of the IM AF files and it plays the components such as audio, image, timed-text and metadata. Figure 2.4 shows the architecture of the IM AF interactive music player with the corresponding decoders/parsers for media data.

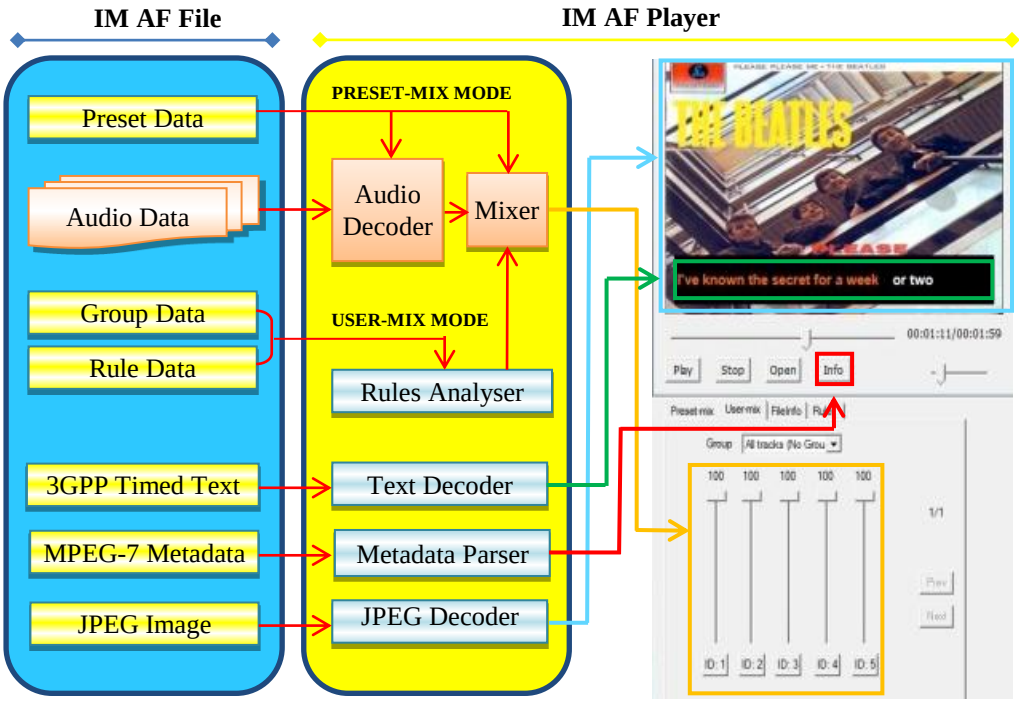

Figure 2.4 - IM AF file format structure and IM AF player architecture with the corresponding media data decoders/parsers.

2.3 Brand identification

In order to identify the specifications to which a file based on ISO base media file format complies, *brands* are used as identifiers in the file format. They are set as four-character codes in a box named *File Type Box* '*ftyp*', which must be placed in the beginning of the file. A brand might indicate the type of encoding used, how the data of each encoding is stored, constraints and extensions that are applied to the file, the compatibility or the intended usage of the file.

A File Type Box contains two kinds of brands. One is '*major_brand*' which identifies the specification of the best use for the file. It is followed by '*minor_version*', an informative 4 bytes integer for the minor version of the major brand. The second kind of brand is '*compatible_brands'*, which identifies multiple specifications to which the file complies.

For the IM AF standard, the brand is related to the maximum number of simultaneously decoded audio tracks and depends on the processing capabilities of the device. Table 2.3 defines brands for each application of IM AF. In all brands, associated data such as JPEG image, 3GPP Timed Text and Metadata based on MPEG-7 MDS are supported and each of them may exist in the IM AF file.

	AUDIO				MAX # OF SIMULTANEOUSLY DECODED AUDIO TRACKS	MAX SAMPLING FREQUENCY/BITS	PROFILE/LEVEL	APPLICATION
BRANDS	AAC	MP3	SAOC	PCM				
'im01'	○	○			4	48kHz/16bits	AAC/level 2	Mobile
'im02'	○	○			6			
'im03'	○	○			8			
'im04'	○	○	○		2		AAC/level 2 SAOC Baseline/2	
'im11'	○	○		○	16		AAC/level 2 SAOC Baseline/3	Normal
'im12'	○	○	○		2			
'im21'	○			○	32	96kHz/24bits	AAC/level 5	High-end

[Remark 1] The audio component data marked as "○" may exist in the file.

[Remark 2] For 'im04' and 'im12', simultaneously decoded audio tracks consist of tracks related to SAOC, which are a downmix signal and SAOC bit stream. The downmix signal shall be encoded using AAC or MP3.

[Remark 3] For all brands, the maximum channel number of each track is restricted to 2 (stereo).

Table 2.3 - Brands for IM AF

Brands starting with '*im0*' are basically supporting the mobile application such as mobile phones and portable audio player. For them, the maximum number of audio tracks to be decoded simultaneously is limited to 4, 6, 8 and 2, respectively, due to the limited processing power of the used devices.

Brands starting with '*im1*' are basically supporting for the general application, such as CD (Compact Disk) and online music service, on more powerful devices than the previous ones for '*im0*', like PC and last generation of mobile devices.

The '*im21*' is the brand for the high-end application for professional user, like Digital Audio Workstation (DAW). In this brand, only AAC and PCM for audio component are supported and the maximum number of audio tracks to be decoded simultaneously is limited to 32.

2.4 Hierarchical structure of audio tracks

IM AF allows several hierarchical levels for the tracks by defining *groups*. Several audio tracks can be gathered in a group (i.e., all guitars of a song). A group can contain tracks or group of tracks. The groups' container is *Group Container Box* '*grco'* box, it hosts one or more *Group Boxes* '*grup'*, as many as the groups desired by the producer. Number of tracks per group, name and number of groups are defined by the producer.

Figure 2.5 - Structure of a song composed of groups

Using the *group_activation_mode* parameter, it is possible to have an additional control on the number of tracks that are in *active state* when a group is played. For instance, a group can contain different versions of the same

instruments in a song, as in Figure 2.5. It is possible to set just one track per each group to be in active state while the song is played, without overlapping the playing of more than one version of the same instrument. This feature is strictly connected to the *Min/Max mixing rule*, and so further details will be given in the respective section of this chapter.

2.5 Preset information

A *preset* stores the volume of each track in a song; more than one preset can be included in an IM AF file. They could be decided by the producer at file creation time or included by the user in a later stage. Presets are involved in the *Preset-mix mode* of the player: this is the fastest way for the user to switch between different mixes of the same song, and it is useful especially on devices with limited interface, such as car-audio and small mp3 player.

The *Preset Container Box* '*prco*' contains fields for general information, such as the number of presets and the default preset ID, which indicates the preset activated at the initial condition without any user interaction. It stores one or more *Preset Box* '*prst*' (i.e. a *default preset* could be the original mix by the music producer; a *karaoke preset* can exclude the vocal track, and so on). Each '*prst*' box contains specific pre-defined mixing information, such as *preset ID*, *track IDs* involved in the preset, the *preset type*, playback *volume gain*, and *preset name*. The *flags* value in each Preset box establishes if the preset is visible and editable by the user; choice is accomplished by the producer. A preset can apply to tracks or objects (single channels in an audio track).

The IM AF standard defines some default presets. These form two main categories: *static* and *dynamic*. Using *static presets*, the encoder sets a fixed

volume to each element involved for the whole duration of the song. In *dynamic presets* the volume of a track (or various tracks, simultaneously) can vary over time. The complete list of the available presets is in Table 2.4.

In the recently published amendment of the standard, *dynamic volume change with approximated volume* and *equalization* functionality has been added for each of the audio tracks [3].

preset_type	**DESCRIPTION**
0	static track volume preset
1	static object volume preset
2	dynamic track volume preset
3	dynamic object volume preset
4	dynamic track approximated volume preset
5	dynamic object approximated volume preset
6	Value reserved
7	Value reserved
8	static track volume preset with EQ
9	static object volume preset with EQ
10	dynamic track volume preset with EQ
11	dynamic object volume preset with EQ
12	dynamic track approximated volume preset with EQ
13	dynamic object approximated volume preset with EQ

Table 2.4 - Presets defined in IM AF standard.

The dynamic volume change can be used to create effects like *fade-in* or *fade out*. To perform this task, IM AF uses an efficient representation that defines volume change by a *time interval*, instead of declaring variations at each sample. Volume changes during the time interval can be represented by a triplet *(a, b, c)*:

a) the starting sample number;
b) the duration of the time interval (number of samples) that the volume change takes place;
c) the new volume level at the end of the time interval.

These parameters are specified in *start_sample_number*, *duration_update* and *end_preset_volume_element* values and they completely define the volume change in a track.

Figure 2.6 shows a volume curve of fading quantized by using the dynamic volume change representation in IM AF. In this way, the required storage space for dynamic presets in the file is significantly low.

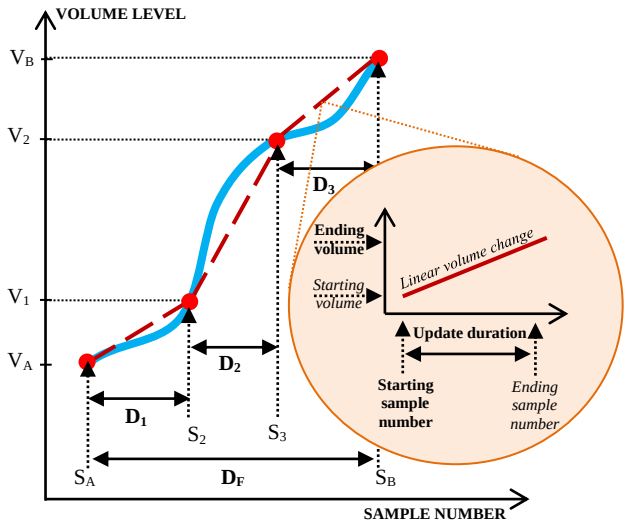

Figure 2.6 - Volume change representation by a time interval. The bolded values are the parameters used by the encoder to define a volume change

Every available preset has a version that provides *equalization* using standard filters implemented into the player: *LPF*, *HPF*, *LSF*, *HSF* and *Peaking*.

More than one filter can be applied on each track and the use of equalization is very easy for the producer: each filter only requires a set of tuneable equalization parameters (*central frequency*, *bandwidth* and *gain*) as shown in Table 2.5; the encoder saves this information into the preset box, then the equalization is performed by the built-in filters in the IM AF player.

FILTER TYPE (filter_type)	FILTER_REFERENCE_FREQUENCY	FILTER_GAIN/ END_FILTER_GAIN	FILTER_BANDWIDTH
LPF (1)	Cut-off frequency (F in Hz)	Undefined	Slope (S in dB/octave)
HPF (2)	Cut-off frequency (F in Hz)	Undefined	Slope (S in dB/octave)
LSF (3)	Corner frequency (F in Hz)	Gain (G in dB)	Slope (S in dB/octave)
HSF (4)	Corner frequency (F in Hz)	Gain (G in dB)	Slope (S in dB/octave)
Peaking (5)	Center frequency (F in Hz)	Gain (G in dB)	Quality factor (Q)

Table 2.5 - Details about equalization in IM AF: available filters and parameters

2.6 Interactivity rules

IM AF standard provides the usage of some restrictions that can be imposed by the producer/artist to limit the changes operated by users on a song, with the aim to preserve the original artistic creation (i.e., the vocal track in a song cannot be muted, or the volume of the guitar must always be higher than the keyboard volume). These restrictions are called *rules*. Their definition is optional and so not imposed by the format.

The *Rules Container Box* '*ruco*' stores one or more rule. Every rule (save some exceptions that will be highlighted) can be applied between two elements identified by their IDs through *element_ID* and *key_element_ID* parameters. Such elements could be either tracks or groups. Therefore, if the producer needs to apply a rule between more than two tracks, group are needed for this purpose.

IM AF defines two kinds of rules which are applied on both selection and mixing of audio tracks, so it is possible to distinguish between *Selection Rules* and *Mixing Rules*.

2.6.1 Selection Rules

The first kind of rule affects the selection of the elements in a song (audio tracks or groups) at rendering time. For instance, the producer could decide that the guitar solo track in a song must not be muted, or that the keyboards and the strings must always not be played in the same time during the song.

***"selection_rule_type"* value**	**SELECTION RULE**
0	Min/Max rule
1	Exclusion rule
2	Not mute rule
3	Implication rule

Table 2.6 - Selection rules in IM AF

The four kinds of selection rules are: *exclusion rule*, *not mute rule, implication rule and min/max rule*. Each one of these essentially defines if a track is in active state or not. Every selection rule is stored in a *Selection Rule Box* '*rusc*'. Details about every selection rule are:

1) **Exclusion rule**: this rule allows specifying that several elements of a song will never be in the active state at the same time, so they will never be played at the same time during the song. This rule can be defined by a *NAND logic operator* between two elements. Thus, A Ɵ B (where Ɵ stands for exclusion and A and B are elements) means that the active state of the element A implies the exclusion of the element B, and vice versa.

2) **Not Mute rule**: it is the simplest rule available. It applies just to one element per time (hence, just *element_ID* needs to be defined), setting a permanent active state for it during the rendering time.

3) **Implication rule**: this rule links the state of one element to another. The two used parameters have the following meaning:
   - *key_element_ID* defines the element that imply the activation;
   - *element_ID* defines the implied element.

   To indicate this rule, the standard uses the symbolism A → B and B → A (where → stands for implication and A and B are elements) and 3 cases to specify how it works:
   - A → B means that if the element A is in the active state, then the element B will be in the active state too;
   - If the element B is in the inactive state then the element A hold its own state;
   - Finally, if the element A is in the inactive state then the element B might be in the active or inactive state.

4) **Min/Max rule**: it applies to groups only, so only the *element_ID* parameter needs to be defined and this must be a group ID. The rule allows specifying both minimum and maximum number of elements of the group that might be in active state. Two extra parameters are used for this rule:
   - *min_num_elements* that specifies the minimum number of elements of the group that might be in active state at the same time;
   - *max_num_elements* that specifies the maximun number of elements of the group that might be in active state at the same time;

   The default minimum and maximum values are, respectively, 0 and n, where n stands for the number of elements contained in the group where rule is applied.

2.6.2 Mixing Rules

The second kind of rule is related to the audio mixing. These rules are also defined by the creator of the IM AF file and will referee the user's changes of groups and audio tracks volumes when the song is played. For instance, the producer could decide that the rhythmic section (i.e., bass and drums) must always have the same volume, or that the volume of the vocal track must always be higher than the other tracks during the song. Mixing rules shall be considered only for the elements which are in active state at rendering time.

***"mixing_rule_type"* value**	**MIXING RULE**
0	Equivalence rule
1	Upper rule
2	Lower rule
3	Limit rule

Table 2.7 - Mixing rules in IM AF

The four kinds of mixing rules are*: equivalence rule, upper* or *lower rule and limits rule*. They are basically used for coupling two elements in a song (except the *limits rule*). Every mixing rule is stored in a Mixing Rule Box '*rumx*' and can be applied between elements in the same group or to elements belonging to different groups. Details about every mixing rule are on the following:

1) **Equivalence rule**: this rule can be applied between two elements in a song. It defines a strict equivalence relationship the volumes of the two elements, so that they are always played at the same volume during the song.

2) **Upper/Lower rules**: these rules define a strict superiority/inferiority relationship between the volumes of the selected elements. *Upper rule* applied between two elements A and B means that the volume of element A (*key_element_ID*) will always be higher than the volume of element B (*element_ID*), and vice versa for the *lower rule*.

3) **Limits rule**: it applies just to one element per time (hence, just *element_ID* needs to be defined), fixing the minimum and maximum limits for the volume of the selected element. Two extra parameters are used for this rule:
   - *min_volume* that specifies the minimum volume of the element;
   - *max_volume* that specified the maximum volume of the element.

2.7 Additional media data

The novelty of IM AF is on additional media data that enrich the user's interaction space, like timed text synchronized with audio tracks, which can represent the lyrics or the chords of a song, and images related to the song, album and artist.

Independently from the chosen mode (*Preset-Mix* or *User-Mix*), the IM AF interactive player can always display these contents while a song is played.

2.7.1 Still picture

IM AF supports the *JPEG* file format for *still pictures* [5]. Pictures can be associated, i.e., to the music album's cover and/or to artist's photos. Images can be the media defined by a '*trak*' box or can be included as metadata in the standalone *Meta Box* '*meta*' in the IM AF file format. Similar to MPEG-7

creation metadata can also be a hierarchy of still pictures at album and song and track level, as will be described in the relative section (Paragraph 2.8).

2.7.2 Timed text

IM AF provides text visualization supporting the *3GPP* standard [6], with possibility to change style and colour of the font and to add visual effects to it. Timed text data are composed of *text samples* and *sample description*. Every sample is a text *string* (list of characters), which is defined in a '*trak*' box with different parameters compared to the one used for the audio tracks.

Text samples can be followed by sample modifier boxes containing information about how the text string should be rendered such as highlighted text for karaoke. Sample description specifies the way text is rendered, its horizontal and vertical justification, its background and foreground colour, font type and size, etc.

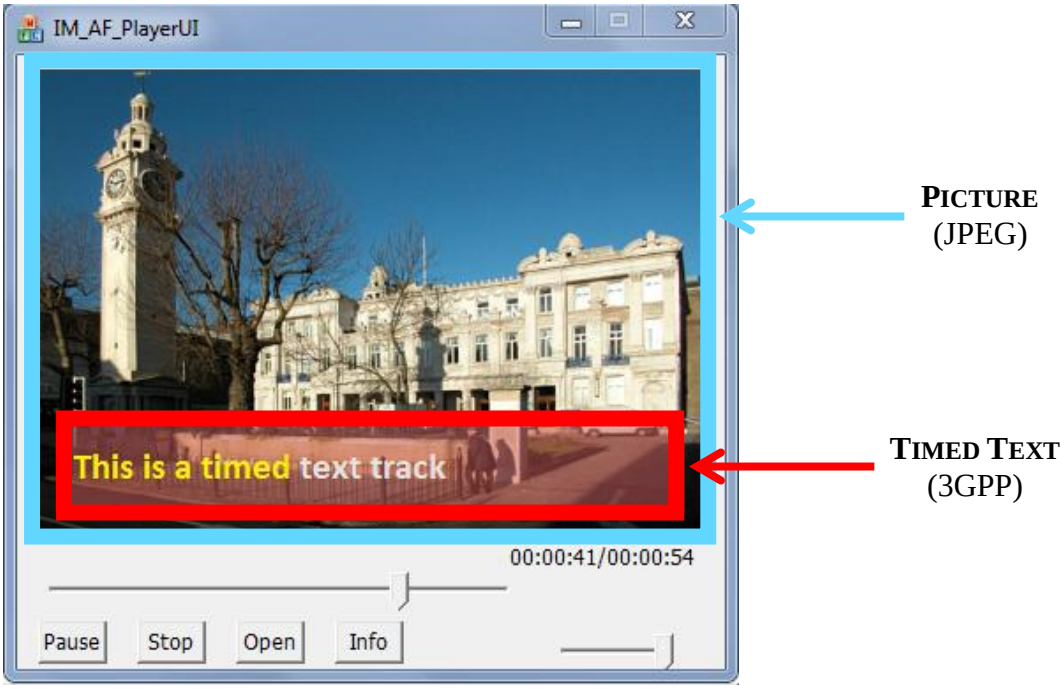

Figure 2.7 - Additional media data in IM AF: picture and timed text

2.8 Metadata

Metadata provides simple background information for a song (i.e. title, singer, album, etc.). In the IM AF file, metadata are stored in the *Meta Box* '*meta*'. Different '*meta*' boxes can be contained at the same time in different levels, as shown in Table 2.8. Every level hosts textual XML metadata information.

METADATA LEVEL	LOCATION
track level	*trak*/*meta* box
song level	*moov*/*meta* box
album level	*meta* box of file

Table 2.8 - Hierarchical levels of metadata in IM AF.

The encoder receives these data either from the producer or extracts them from the *ID3 tags* of the input MP3 files, then stores them all in the *XML Container Box* '*xml*'. *MPEG-7* is the multimedia content description standard supported by IM AF [7].

2.9 Compatibility with legacy players

For legacy players or devices that do not provide multitrack support, IM AF files can still be played. An audio track containing the classic producer's mix of all the instruments in a song (*All Recorded track*, AR) can be included and set as the only "track enabled" that will be played by the legacy player; whereas the IM AF player is able to switch it off and play the multi-track version. In order to distinguish the AR audio track from the audio tracks for interactive music service, *flag* field of *Track Header Box* '*tkhd*' is used. For the AR audio track, the flag is set as '*Track_enabled*' whose value is 0x000001, whereas the flag of the audio track for interactive music service is set as '*Track_disabled*' whose value is zero.

CHAPTER III

THE IM AF ENCODER

Introduction

After seeing into details the specifications of the IM AF standard, we can talk about the implementation of the features for the IM AF encoder.

A first version was proposed, as proof of concept, by the Centre for Digital Music of the Queen Mary University of London, in late August 2012 [26]. It essentially defined the basic structure of an IM AF file, including individual music-tracks encoded in MP3 format, with a limited choice for the presets and the rules. Many features were not integrated and left for future developments. This work comes to fill in this gap, improving the capabilities of the IM AF Encoder and making it as more as possible compliant to the standard.

Beside the definition of the whole set of presets and rules, the main work on the encoder has been done for the inclusion of pictures and timed text data as karaoke lyrics, to be shown during the playing of a song in IM AF format.

The IM AF Encoder uses a simple command line interface and has been developed in C programming language (ANSI C - C89, to be more precise) using Visual Studio 2012.

3.1 Useful tools: Mp4 Explorer and Mp4 Browser

The implementation of a compliant IM AF encoder is mainly based around "filling in" the ISO-BMFF boxes in the right way. The size of the boxes is the most important parameter to be sure of having a playable IM AF file.

Declaring the wrong size of a single box may lead to bytes misalignment and malfunction in the file, even in the case of a single byte error.

Some tools have been extremely useful in the development of a successfully compliant IM AF encoder to the standard. The MP4 Browser [36] and MP4 Explorer tools [37] are free software for Windows that analyse the structure of ISO-BMFF files, showing the content, type and size of all boxes, except the ones introduced and used exclusively by the IM AF file format (like presets and rules boxes) since they were created to work with ISO/IEC 14496 files (MPEG-4).

Without this type of utilities, building an IM AF encoder would be much more difficult.

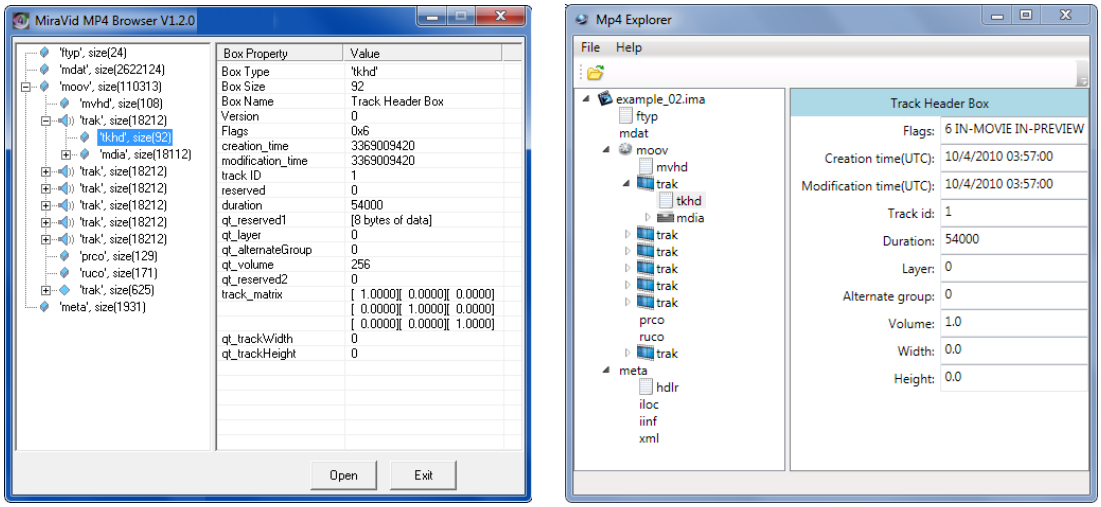

Figure 3.1 - MP4 Browser and MP4 Explorer tools

3.2 Brand definition

The standard's amendment n.2 [2] includes the reference software '***IM AF Player***' and ***6 conformance files***, each one including different features. Since the main target of the work is the inclusion of the ***picture*** and ***timed text*** support for the encoder, the *conformance file n.2* has been considered as model; details about the considered file are in Table 3.1. It has 6 audio tracks and its major brand is '*im02*', but it also covers brands '*im01*', '*im03*', '*im11*' and '*im21*' with AAC audio tracks.

FILE NAME	BRAND (COVERED BRAND)	# OF AUDIO TRACKS	COMPONENTS	PRESET	# OF GROUPS	RULE	
						SELECTION	MIXING
example_02.ima	im02 (im01, im03, im11, im21)	6	- AAC - JPEG - MDS	- Static track volume preset	-	- Implication rule	- Equivalence rule - Limit rule

Table 3.1 - Details of the Conformance File n.2, considered as the model for the inclusion of *picture* and *timed text*.

For the proposed encoder, the MP3 format has been chosen for the audio tracks, with a maximum limit of *6 audio tracks*, for being easily played on any kind of device -- from budget smartphones to the more powerful Digital Audio Workstation (DAW) -- and for an easy inclusion of the encoder in *Sonic Visualiser* at a later stage (see Chapter 5).

The limit is defined by a global variable '*maxtracks*' and can be easily increased. Hence, the created conformance files will cover other brands (according to the Table 2.3) that are '*im03*' and '*im11*'. The major brand remains '*im02*'.

The input files (MP3s, 3GP file and JPEG picture) location is declared on the top of the header *IM_AF_Encoder.h* file.

3.3 Multitrack Audio

The inclusion of multiple audio tracks in the creation of an IM AF file was mainly developed in the early version of the proposed encoder. Therefore, this paragraph will be briefly described, while the full specifications can be found in the referenced work [26].

The first step in the encoding process is to extract the audio samples of every single input MP3 audio file and store them into an ISO-BMFF *Media Data Container Box* '*mdat*'.

Basically an MP3 file is split into small blocks or *frames* where data is stored. Each frame has *1152 samples*. The IM AF Encoder uses ***MP3*** *format* with *constant bit rate* of ***128 kbps***, ***16-bits*** *resolution* and ***44.1kHz*** *sampling frequency*, ***joint stereo*** *mode*.

Each frame has duration of

$$1152/44100 \approx \mathbf{0.026\ sec}$$

and the formula for counting frame length in bytes [43] gives

$$144 \times (\text{bitrate} / \text{sample rate}) + \text{pad} = 144 \times (128000/44100) + 0 = \mathbf{417\ Bytes}$$

where the result has been rounded down.

At beginning and end, MP3 files may also include auxiliary data such as *ID3 TAGs*, metadata with information related to the song, album and artist. The encoder opens each MP3 file and search for the first frame avoiding the *ID3 header*, as it does not contain useful information for the current purpose. Then, the encoder reads any of the audio samples until the end of file and writes them in the '*mdat*' box in output file.

The second step in the encoding process is to extract the following values from each MP3 frame: *size* (in bytes) and *length* (in milliseconds) of the frame, *number of samples* and *duration*. The IM AF encoder writes this information in the corresponding container inside a *Track Box* '*trak*' as it is illustrated in Figure 3.2.

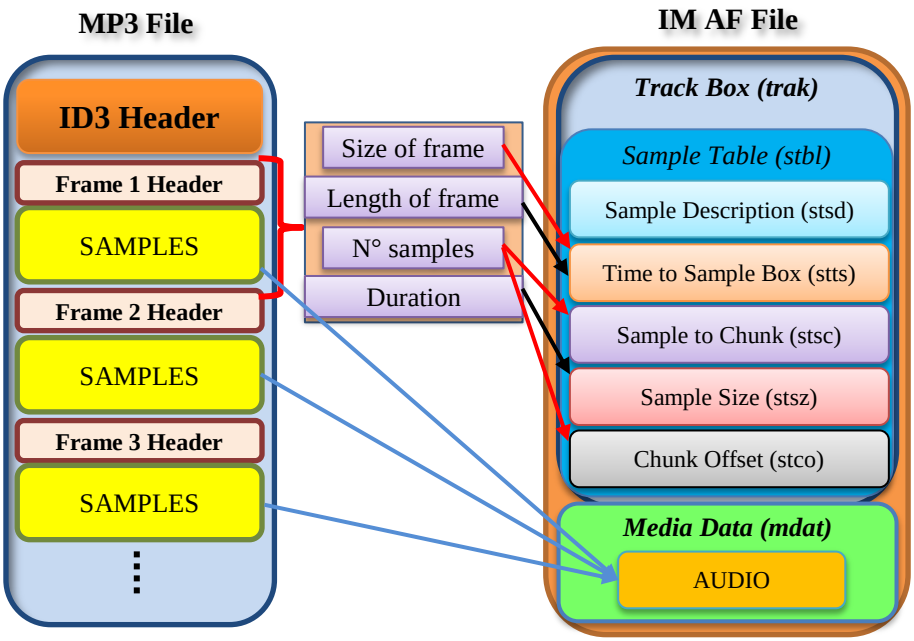

Figure 3.2 - Audio samples and associated data transfer from MP3 to IM AF.

The information needed by the encoder in order to reproduce the media data will be contained in *tables*. These tables are held in boxes inside the *Sample*

Table Box '*stbl*'. The *Sample Description Box* '*stsd*' gives information of the coding type used, the sampling rate and decoder specific information.
The *Time to Sample Box* '*stts*' contains a table that stores for each entry two values: the number of MP3 frames with the same duration and the duration (in milliseconds) of each of these frames. Samples within the media data are grouped into *chunks*.

The *Sample to Chunk Box* '*stsc*' defines a table that includes the size of each chunk and the number of chunks that have the same size. For constant bitrate MP3s, this table includes only one entry, that is, the number of MP3 frames in the chunk since all the chunks are equally sized. In the *Sample Size Box* '*stsz*' are stored the number of MP3 frames once more and a table giving the size (in bytes) of each of these audio frames.

Finally, the *Chunk Offset Box* '*stco*' indicates the position of the beginning of the data of the audio track in the *Media Data Container Box* '*mdat*' of the IM AF file.

3.4 Groups

In IM AF file, the groups hold the *IDs* (*element_ID*) of the contained elements. Since a group contain audio tracks and groups, *element_ID* shall be the ID of tracks (*track_ID*) and/or ID of groups (*group_ID*). Elements IDs are all *32-bits unsigned integer* values.

A *track_ID* shall be represented using values from 0x00000001 to 0x7FFFFFFF and a *group_ID* shall be represented from 0x80000000 to 0xFFFFFFFF. Hence, elements identified by *element_ID* are recognized as a

track_ID or as a *group_ID* according to the Most Significant Bit (MSB) of the element ID. The maximum number of groups allowed by the encoder in an IM AF file is set with the global variable *maxgroups*.

3.5 Presets

The IM AF standard defines some default *presets*, as described in Paragraph 2.5. Despite the availability of 11 presets, only ***6*** of them are really available, as the other are not yet implemented in the reference software. Moreover, the dynamic presets corresponding to the *preset_type* values of "2" and "3" had been not included in any of the provided conformance files. Hence, it is not possible to test completely the functionality of these missing presets when included in an IM AF file. The following Table 3.2 resumes the actual available presets.

preset_type	AVAILABILITY	DESCRIPTION
0	✓	static track volume preset
1	✓	static object volume preset
2	✓	dynamic track volume preset
3	✓	dynamic object volume preset
4	✓	dynamic track approximated volume preset
5	×	dynamic object approximated volume preset
6	-	Value reserved
7	-	Value reserved
8	✓	static track volume preset with EQ
9	×	static object volume preset with EQ
10	×	dynamic track volume preset with EQ
11	×	dynamic object volume preset with EQ
12	×	dynamic track approximated volume preset with EQ
13	×	dynamic object approximated volume preset with EQ

Table 3.2 - The presets list defined by the IM AF standard and their effective existence in the reference software.

Every preset form a *PresetBox* '*prst*' in the *PresetContainer Box* '*prco*'. The encoder ask user for the number and the kind of desired presets. The maximun number of presets to be included in an IM AF file is fixed to ***10*** by the global variable *maxpresets*. The *preset_ID* value is automatically e progressively assigned, starting from 1. The first declared preset will be the *default preset*, that is the one played by default when the player's Preset-Mix mode is chosen. The *flags* value in each Preset box is set to 0x02 (enable Display, disable Edit); the possibility of using the value 0x03 (enable Display, enable Edit) has still to be explored, since the IM AF player interface does not allow to edit presets yet.

A preset supplied by the encoder involves all the available audio tracks in a song (*num_preset_elements = totaltracks*) and has a fixed overall volume (*preset_global_volume = 100*); both values can be easily changed and maybe asked as input parameters, even it is not clear what happens to the exluded tracks in a preset when *num_preset_elements < totaltracks*.

Presets have different ways from each other to be included in the IM AF file. The function called *presetcontainer()* controls this process, using a *switch/case* structure to select the appropriate implementation, with *preset_type* value as switching variable.

The *static* presets basically require the definition of the playback volume gain for each audio *track* or *object* (channel), by setting the *preset_volume_element* with the desired value and using the quantization Table 3.3 below.

Index	**0**	**1**	**2**	**3**	**…**	**199**	**200**
value (ratio)	0	0.02	0.04	0.06	…	3.98	4.00

Table 3.3 - The quantization table for playback volume gain.

The presets that involve objects require the definition of the *output_channel_type*, that is the number of channels of each included audio track. The IM AF encoder uses *Joint Stereo* tracks only, so the value is set to 1 (for generic stereo tracks). To constraint this, the global variable *num_ch* is used and set to 2.

The only one working dynamic preset is the "*Dynamic track approximated volume*". It can be used to create effect like *fade-in* and *fade-out* and it uses the dynamic volume change representation described in Paragraph 2.5. The maximum number of volume changes in a track is set by the global variable *maxdynamic*.

The *start_sample_number* and *duration_update* are integer values that indicate the sample where the volume change takes place and the number of samples that it occurs, respectively. It can be possible to refer to the playing time rather than the samples by multiplying both values for *0.026* (see Paragraph 3.2).

Finally, the only one working preset with equalization "*Static track volume with Equalization*". Equalization can be applied to every track individually; more than one filter per time on each track can be used. The maximum number of filters that is possible to apply to every track is set by the global variable *maxfilters*.

Built-in IM AF player filters can be used for this purpose and their use is desribed in Paragraph 2.5 and in details in Amendment 3 [3]. The filters are based on a 2^{nd} order IIR structure.

Defining the parameters of a filter could be problematic without an user interface, for a not-expert user.

To make it easy to use, only one filter has been set in the IM AF encoder, with the following parameters:

- *filter_type* = 4 (HPF)
- *filter_reference_frequency* = 5000 (10kHz)
- *filter_gain* = N.D. for HPF (G = *filter_gain*/5-41 [dB])
- *filter_bandwidth* = 4 (S = *filter_bandwidth* × 6 = 24 dB/octave)

3.6 Rules

Every preset defined by the standard specification is included and fully working in the developed encoder. As described in Paragraph 2.6, rules are from two categories: Selection Rules and Mixing Rules.

The IM AF encoder needs specific information for each one of the rule; required parameters are briefly summarized in the following tables:

SELECTION RULES						
Type	**mixing_rule type**	**element ID**	**key_element ID**	**min_num elements**	**max_num elements**	**rule_description**
Min/Max	0	Group ID required	-	✓	✓	✓
Exclusion	1	✓	✓	-	-	✓
Not mute	2	✓	-	-	-	✓
Implication	3	✓	✓	-	-	✓

Table 3.4 – Selection rules in IM AF.

MIXING RULES						
Type	**selection_rule type**	**element_ID**	**key_elem_ID**	**min volume**	**max volume**	**rule_description**
Equivalence	0	✓	✓	-	-	✓
Upper	1	✓	✓	-	-	✓
Lower	2	✓	✓	-	-	✓
Limit	3	✓	-	✓	✓	✓

Table 3.5 – Mixing rules in IM AF.

The function *rulescontainer()* defines the structure of the *Rule Container Box* '*ruco*', which hosts one or more *rule boxes* (*Selection Rule Box* '*rusc*' or *Mixing Rule Box* '*rumx*').

Every rule in the developed encoder, for now, applies only to couple of audio tracks. In general, elements involved in a rule could be either audio tracks or groups of them. Moreover, the encoder allows the presence of only one rule for each category (one for selection and one for mixing).

3.7 JPEG Still Pictures

A picture is included as metadata in the standalone *Meta Box* '*meta*' in the IM AF file format. The size of the desired JPEG picture is calculated by the function *imagesize()*, then the whole *'.jpeg*' image file is copied in the Media *Data Container Box* '*mdat*' box using the function *insertimage()*.

The image's *size* and its *offset* in the '*mdat*' are saved in the *Item Location Box* '*iloc*', in particular into the '*extent_lenght*' and '*extent_offset*' values. Other

information, such as picture's name and encoding type, are stored in *Item Information Box* '*iinf*'.

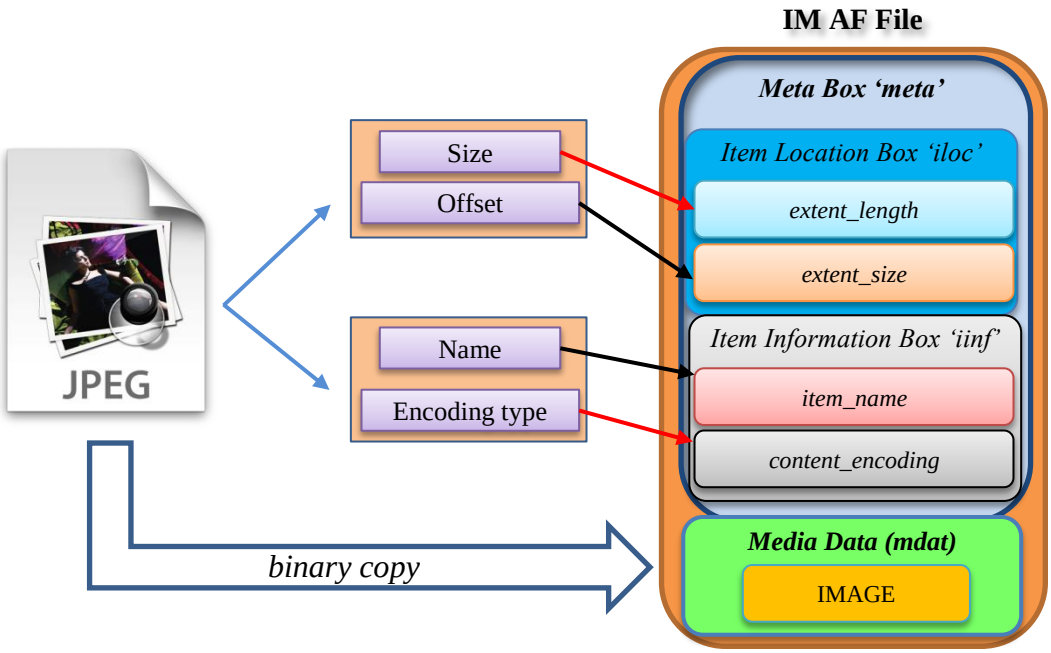

Figure 3.3 - JPEG picture inclusion process in IM AF Encoder

3.8 3GPP Timed Text

The developed encoder is able to include a timed text track in an IM AF file. Text samples (strings) and their description are imported from a 3GP file, created using external software, like QuickTime Player [22]. Information about the strings and their display start/end time can be initially included in a text file, using the following syntax:

[start time] *Text String* **[end time]**

This '*.txt*' file can be opened and then exported in 3GP format using QuickTime Player. An ISO-BFFF structure will be created, with a '*mdat*' box

that contains the text strings and '*stts*' box with information about the text timing. Storage of timing information is similar to audio tracks, as the same boxes are used ('*stts*', '*stsc*', '*stsz*' and '*stco*' as showed in Figure 3.2).

Duration (in samples) for every string is saved in '*stts*' box according to the following relation:

$$sample\ delta = duration\ of\ interval\ (sec) \times timescale \qquad [3.1]$$

where *timescale* is an integer value that indicates the time unit per second. The default value is 600, which is the normal movie time scale [22].

Example 3.1 - A set of 3 phrases is included in the text file, with the following syntax:

[00:00:00.000] *String A* **[00:00:02.000]**
[00:00:02.000] *String B* **[00:00:05.000]**
[00:00:05.000] *String C* **[00:00:10.000]**

that stands for the displaying of the text "*String A*" between seconds 0 and 2, "*String B*" between seconds 2 and 5 and "*String C*" between seconds 5 and 10.

QuickTime creates 3 entries in '*stts*', where the 3 "*delta*" values are obtained using the relation [3.1]:

(count, delta)[0] = (1,1200)	->	*1200 = 2 seconds x 600*
(count, delta)[1] = (1,1800)	->	*1800 = 3 seconds x 600*
(count, delta)[2] = (1,3000)	->	*3000 = 5 seconds x 600*

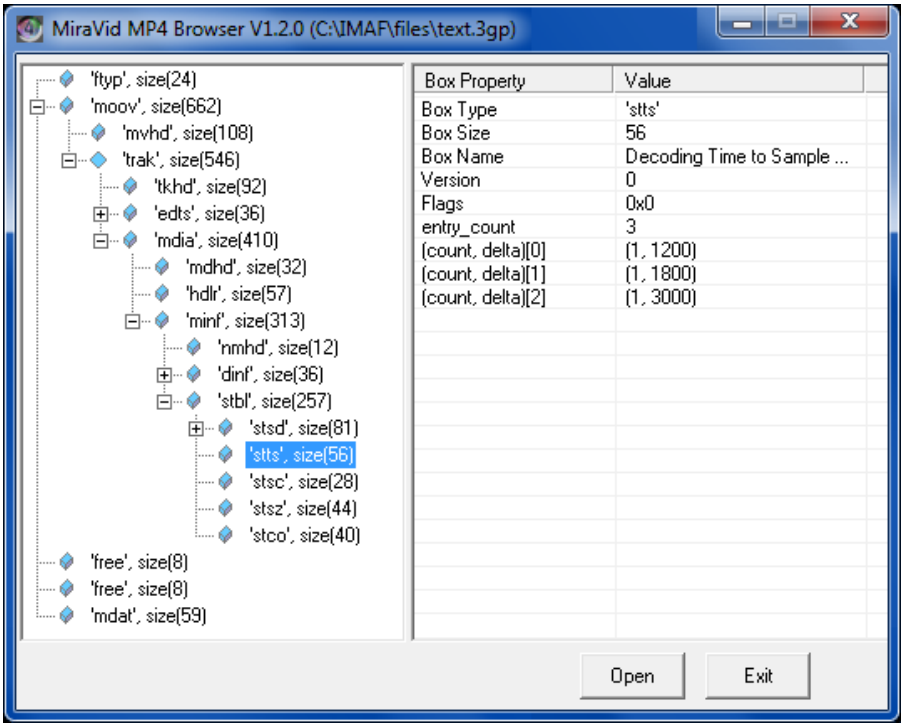

Figure 3.4 - Structure of the 3GPP file in Example 3.1. The figure highlights the content of the '*stts*' box.

The '*mdat*' and '*stts*' are the only two boxes contained in 3GP file that the encoder uses to create the timed text track. Boxes like 'stsz', 'stco', 'stsd' and 'stsc' are created from scratch in the IM AF file.

The encoder seeks every text string inside the text file and it copies them in the '*mdat*' box of IM AF file. For the encoder purpose, the timed text track is used to display the lyrics of the played song, so it requires some modifiers to be rendered in *karaoke style*. In '*stsz*' is stored the size of every string, including the modifiers like the highlight colour of the text or the highlight start/end time of words in a phrase.

Another important box is '*tx3g*', contained inside *Sample Description Box* '*stsd*', which defines sample descriptions for the text track (i.e. font type and size, horizontal and vertical justification, background colour) [6].

3.9 Metadata

The container for metadata information is *Meta Box* '*meta*'. This is required to contain a *Handler Box* '*hdlr*' indicating the structure or format of the contents. MPEG-7 streams, which are a specific kind of metadata stream, have their own handler declared, documented in the MP4 file format [7].

For IM AF a generic metadata track handler is sufficient, so *handler_type* is simply set to '*meta*' and the *name* value, which is a string that gives a human-readable name to the box for debugging and inspection purposes, is set to '*mp7t*'.

The '*meta*' box purpose is twofold: it hosts the descriptive or annotative *metadata* (about song, album, artist, etc.) and provides information about the JPEG picture included in the IM AF file by locating its *offset* within the file and its *length* (in bytes).

The metadata is located in the *XML Box* '*xml*' in XML format, as the name implies. The IM AF encoder simply includes such information in the resulting file by writing the *string* value. For now, this value can be changed in the source code only.

The *Item Location Box* '*iloc*' provides information about the position within the file of different resources. In the IM AF encoder, it is only used to locate the JPEG picture store in the *Media Data Container* '*mdat*', so the *item_count* is set to 1. The *extent_offset* parameter provides the absolute offset in bytes from the beginning of the IM AF file, while *extent_length* provides the absolute length in bytes of the picture.

The *Item Information Box* '*iinf*' provides extra information about selected items located by 'iloc', like *item_name*, *content_type* and *content_encoding*.

This information is not mandatory and there is no feedback provided by the IM AF player about them. They are just used for debugging and inspection purposes.

3.10 Software Version Control

Working in a team on a project like the IM AF Encoder, requires a tool that allows the easy consultation, modification and exchange of the code for every developer. For this purpose, we made use of Sound Software and Easy Mercurial.

Sound Software [38] is an online file repository based on Mercurial [39], a free, distributed CVS (Concurrent Versions System). It efficiently handles projects of any size and offers an easy and intuitive interface, allowing team members to track the history of their work and to help collaborate each other.

Easy Mercurial [40] is an interface for the Mercurial system and allows easy downloading (pull) and uploading (*push*) of files on the repository.

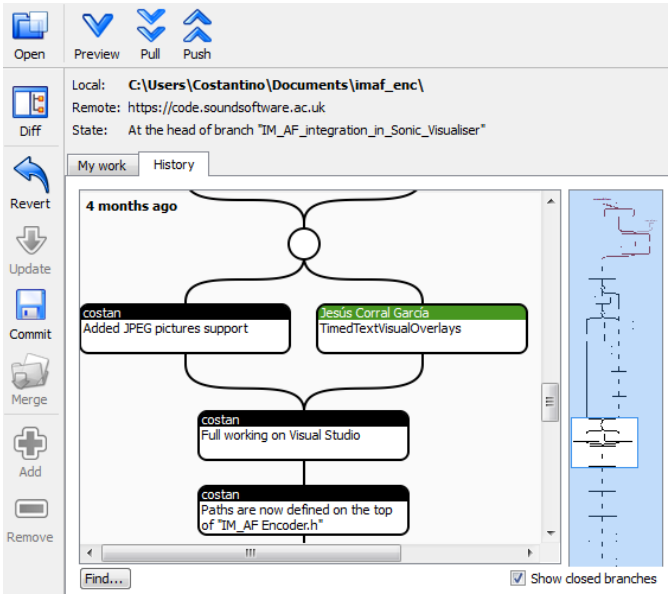

Figure 3.5 - Updating the Sound Software repository with Easy Mercurial

CHAPTER IV

RESULTS

Introduction

The capabilities of the developed encoder are tested by creating some conformance files. Conformance files shall be readable by the IM AF reference software and used to verify some possible combinations of features defined by the specification of the IM AF.

A comparison between the created conformance file will be highlighted by defining some conformance point. Every point shall contain specific supported components.

4.1 Creation of an IM AF file

When the IM AF encoder is executed, it is possible to create an IM AF file following the instructions for every step on the screen. In Chapter 3 all the required parameters for every feature are described.

Table 4.1 lists these parameters, indicating the ones that are required as input from the user and the few ones that are automatically set.

4.2 Conformance points

An IM AF file that complies with the standard must conform to some conformance points. These points are like subcategories that arrange the files by their supported components. Conformance points must not be confused with brands: in all brands, components such as image, timed text and metadata may exist. The conformance points of Interactive Music AF (IM AF) are followed:

- ***Conformance point 1 (C1)*** provides basic capability to play mixed audio track according to the preset. An IM AF file that conforms to this conformance point shall have at least one *preset information* (at least one '*prst*' box). Furthermore, the file shall contain the following supported components:
  - *Interactive Music AF compatible to ISO Base Media File Format*
  - *One of the following supported audio components:*
    - ✓ MPEG-1 Audio Layer III
    - ✓ MPEG-4 Audio AAC profile
    - ✓ MPEG-D SAOC Baseline profile
    - ✓ PCM

PARAMETER	INPUT/CHOICE FROM USER	SET IN THE CODE (USED PARAMETER)
FILES FOLDER		*files_path*
➢ JPEG picture	✓	
└> Image file name		*image_path*
➢ 3GP file for timed text		Mandatory
└> Text file name	✓	
AUDIO TRACKS		
➢ Max number of audio tracks		*maxtracks*
➢ Number of audio tracks per file	✓	
└> Name of audio tracks	✓	
GROUPS		
➢ Max number of groups		*maxgroups*
➢ Number of groups per file	✓	
➢ Name of the group	✓	
➢ Description	✓	
➢ Number of elements	✓	
└> Involved elements IDs (Tracks only)	✓	
➢ Activation mode	✓	
➢ Volume	✓	
PRESETS		
➢ Max number of presets		*maxpresets*
➢ Number of presets per file	✓	
└> Type	✓	
└> Tracks volume	✓	
└> Number of dynamic volume changes		1
└> Start sample		100 (* 0.026s = *2.6 seconds*)
└> Change duration		500 (* 0.026s = *13 seconds*)
└> End volume		fixed for *fade in/out*
└> Equalization on the track	✓	
EQUALIZATION		
➢ Number of filters per track		1
└> Type		HPF
└> Central frequency		5000 (10kHz)
└> Gain		*N.D. for HPF* (G=Gain/5-41 (dB))
└> Bandwidth		4 (S=Bandwidth*6 (dB/octave))
RULES		
SELECTION RULES		
➢ Number of Selection Rules		1
└> Type	✓	
└> Element ID (Track A ID)	✓	
└> Key Element ID (Track B)	✓	
└> Min elements number	✓	
└> Max elements number	✓	
MIXING RULES		
➢ Number of Mixing Rules		1
└> Type	✓	
└> Element ID (Track A ID)	✓	
└> Key Element ID (Track B ID)	✓	

Table 4.1 - Detailed list of required parameters for the IM AF Encoder.

- ***Conformance point 2 (C2)*** provides basic capability to play mixed audio track according to the user's interactive settings (such as group/track selection and volume control) which comply with the rule. In addition to the capabilities of conformance point 1, an IM AF file that conforms to this conformance point shall have at least one *group information* (at least one '*grup*' box) and one *rule information* (at least one '*rusc*' or '*rumx*' box). Files that conform to this conformance point shall contain the following supported components:
  - *The components specified for conformance point 1*

- ***Conformance point 3 (C3)*** provides capability to play *image*, *timed text* and *metadata* in addition to the capabilities of conformance point 1. Files that conform to this conformance point shall contain the following supported components:
  - *The components specified for conformance point 1*
  - *JPEG*
  - *3GPP Timed Text*
  - *MPEG-7 Multimedia Description Scheme*

4.3 Conformance files

Compliance of the encoder to the IM AF standard is achieved by creating IM AF files, similar to conformance files provided by the standard. Every conformance file belongs to the '*im02*' brand, defined in Paragraph 2.3.

For the sake of completeness, four conformance files are compared: one for each of the three Conformance Point and one with all the possible features that an IM AF file can include. For this work, compliance to Conformance Points 2 and 3 only need to be checked, as compliance to Conformance Point 1 was

already proved by the first encoder implementation [26]. Moreover, since the inclusion of a timed text track is mandatory for the last version of the proposed encoder, as shown in Table 4.1, compliance to Conformance Point 2 is checked by using an IM AF file previously created with an earlier version of the code.

	FILE STRUCTURE						AUDIO				METADATA	IMAGE	TEXT
N.	**ISO-BMFF**	**Brand**	**'grco'**	**'prst'**	**'rusc'**	**'rumx'**	**MP3**	**AAC**	**PCM**	**SAOC**	**MDS**	**JPEG**	**3GPP TT**
1	C1	C1	-	C1	-	-	C1	-	-	-	-	-	-
2	C1	C1	C2	C1	C2	C2	C1	-	-	-	-	-	-
3	C1	C1	-	C1	-	-	C1	-	-	-	C3	C3	C3
4	C1	C1	C2	C1	C2	C2	C1	-	-	-	C3	C3	C3

Table 4.1 - Compliance of the created IM AF files with conformance points**.**

N.	FILE NAME	BRAND (COVERED BRAND)	# OF AUDIO TRACKS	COMPONENTS	PRESETS	# OF GROUPS	RULES	
							Selection	Mixing
1	*CF01.ima*	im02 (im03, im11)	6	- MP3	- Static track volume preset	-	-	-
2	*CF02.ima*	im02 (im03, im11)	6	- MP3	- Static track volume preset	1	Min/Max rule	Limit rule
3	*CF03.ima*	im02 (im03, im11)	5	- MP3 - MDS - JPEG - 3GPP	- Static track volume preset - Dynamic track volume preset	-	Exclusion rule	Upper rule
4	*CF04.ima*	im02 (im03, im11)	5	- MP3 - MDS - JPEG - 3GPP	- Static track volume preset - Dynamic track volume preset - Static track volume preset with EQ	2	Not mute rule	Equivalence rule
[Note] Major brand of all the conformance files is 'im02'; it can be also 'im03' and 'im11' because difference between them is only *the maximum number of simultaneously decoded audio tracks* and it depends on the *player performance*. Hence conformance files cover the conformance test for 'im03' and 'im11' with MP3 audio tracks.								

Table 4.2 - Details of the created IM AF conformance files.

The IM AF player parses and plays the IM AF files and playing the components such as audio, image, timed-text and metadata. Especially for audio, multiple audio tracks are mixed according to the preset parameters or user's direct selection/control that conform to the rules made by a producer.

4.4 Playing the files

Every created IM AF file has been parsed and played successfully using the reference software decoder. The IM AF player parses the IM AF files and plays the components such as audio, image, timed-text and metadata, as shown in Figure 4.1.

Especially for audio, multiple audio tracks are mixed according to the preset parameters (in Preset-Mix mode) or user's direct selection/control that conforms to the rules (in User-Mix mode).

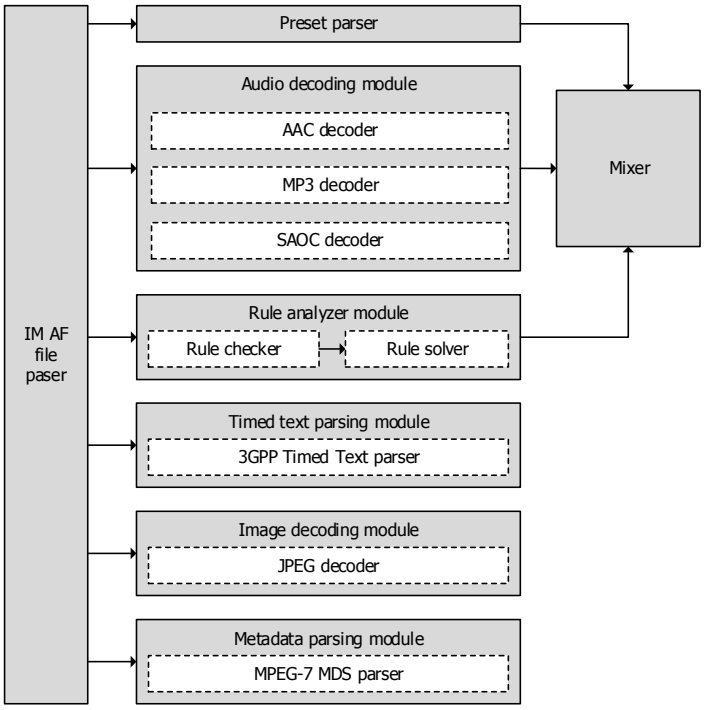

Figure 4.1 - The architecture of the IM AF player.

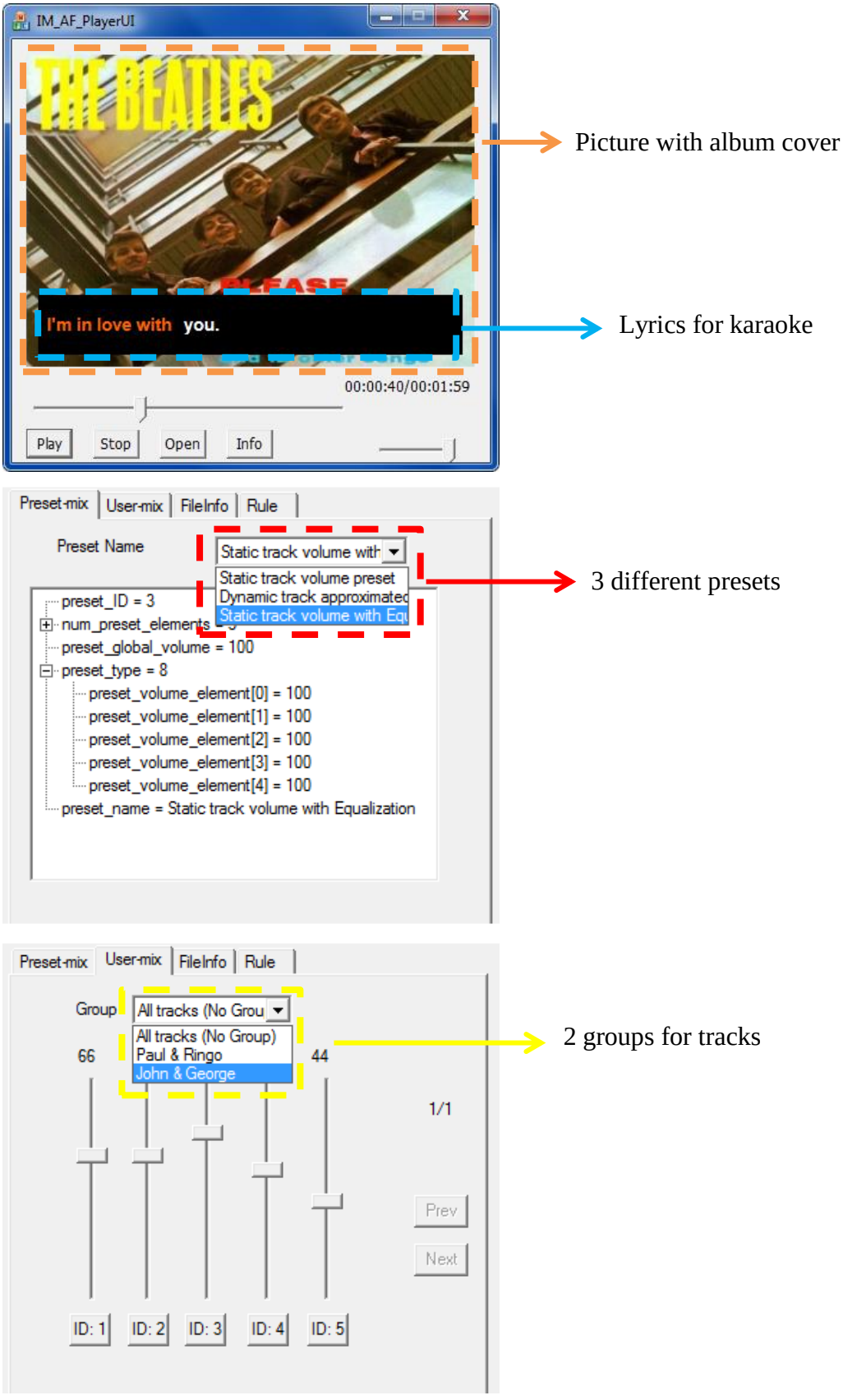

Figure 4.2 - Details about the created CM_04.ima file (*picture*, *lyrics*, *presets*, *groups*).

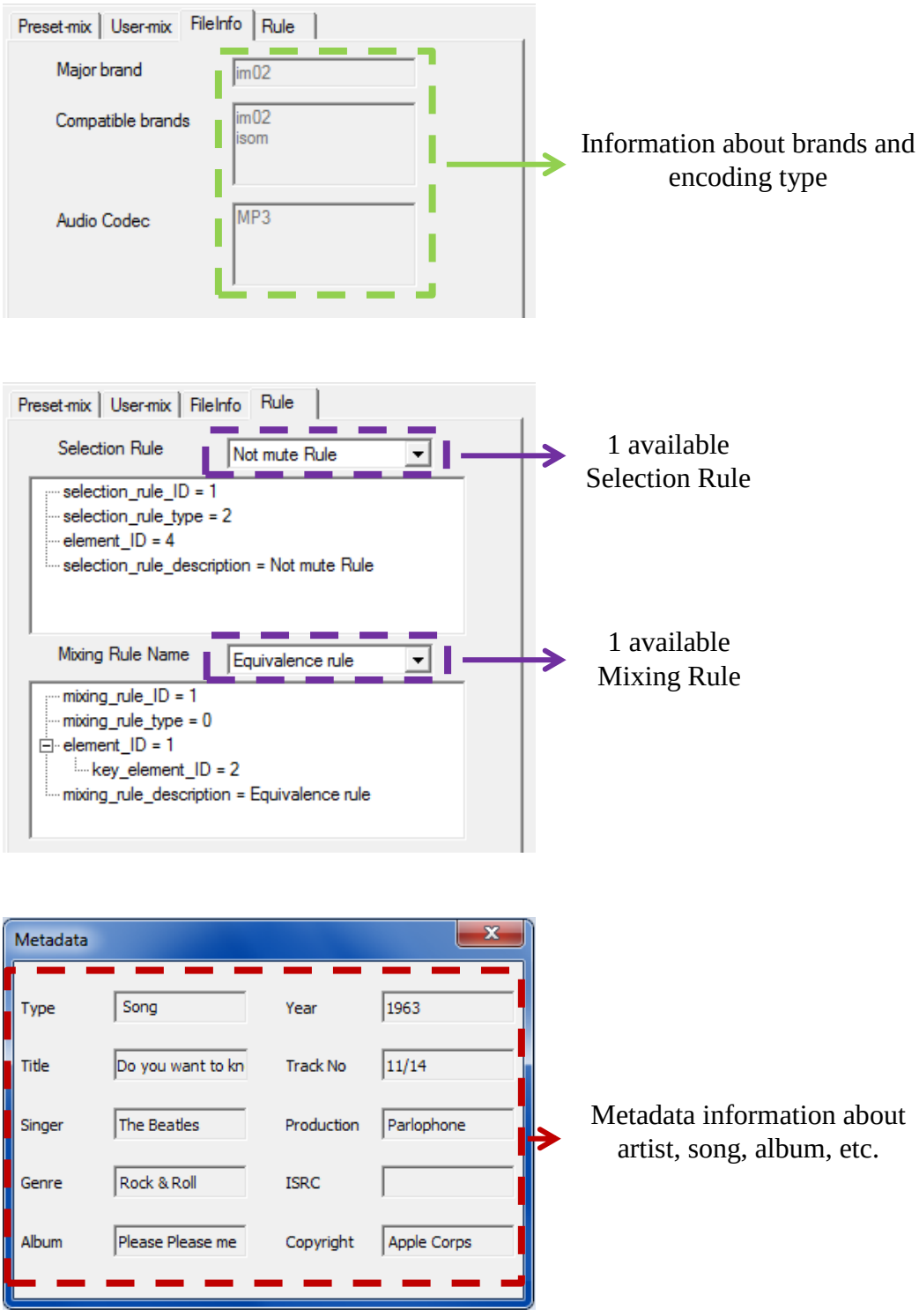

Figure 4.3 - Details about the created CM_04.ima file (*file info*, *rules*, *metadata*).

CHAPTER V

INTEGRATION IN SONIC VISUALISER AND FUTURE DEVELOPMENTS

Introduction

Once the encoder's compliance to the IM AF standard has been verified, we can focus on the second goal of this work, which is the integration of the IM AF Encoder in Sonic Visualiser.

Sonic Visualiser is a friendly and flexible end-user desktop application for analysis, visualisation and annotation of music audio files. Providing IM AF support for this software can open new horizons to the development of new audio tools and to the music research in general, as it will be presented with some use-case scenarios.

In this work, only the possibility of creating IM AF files has been provided in Sonic Visualiser. This latter is not yet able to play IM AF files, since the inclusion of the reference software player requires importing several libraries in the already vast project that Sonic Visualiser is. Therefore, this part is beyond the scope of this work and it is left as a possible future development.

5.1 Introduction to Sonic Visualiser and VAMP plugins

Sonic Visualiser is an open-source, cross-platform application designed to assist in a friendly way the study and comprehension of what lies inside an audio file, with particular emphasis on musical recordings [9]. It loads audio files in WAV, Ogg and MP3 formats, and it displays their waveforms. It allows audio visualisation such as spectrogram views, with interactive adjustment of display parameters, and also to add overlay annotations on top layers. The architecture of the application allows importing several tracks in a session and to edit or apply effects to each one of them individually.

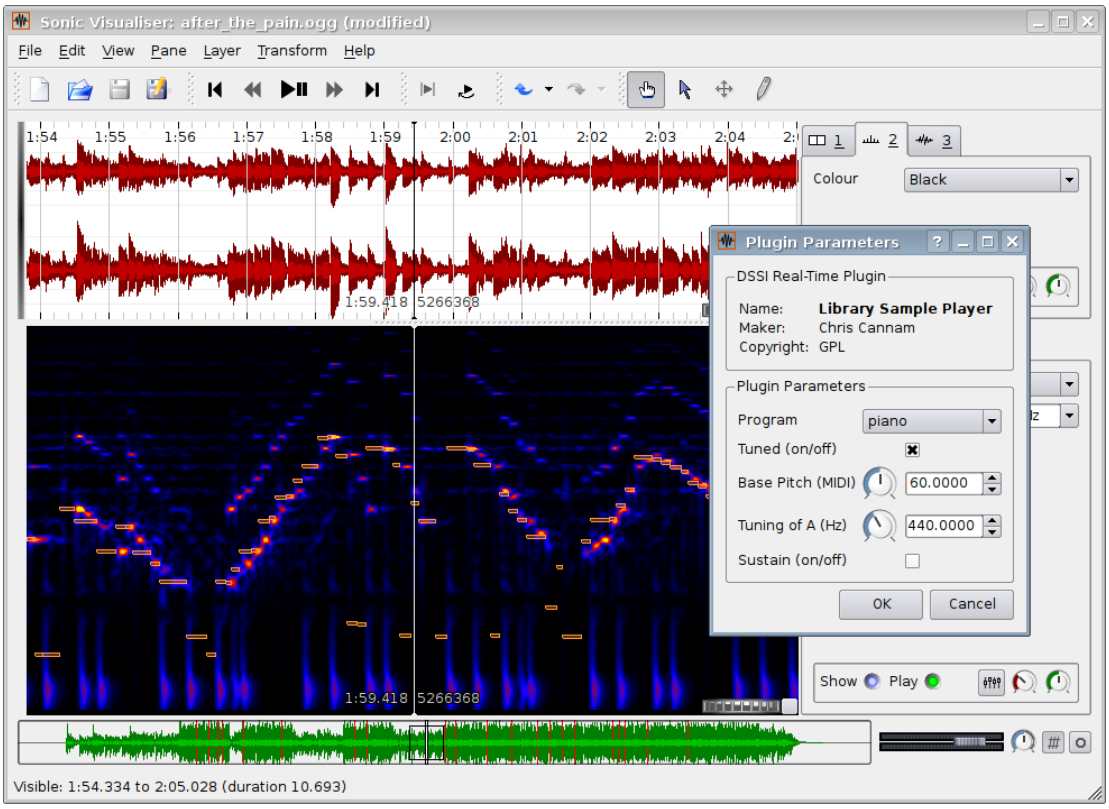

Figure 5.1 - Sonic Visualiser user interface.

An essential strength of Sonic Visualiser is its ability to support third-party plugins. The native plugin format is called *Vamp*, an audio processing system for plugins designed for audio features extraction [41].

Different from a VST (*Virtual Studio Technology*) plugin, a Vamp plugin is not designed to generate sounds and effects, but to recover symbolic information from an audio file, like the key and the chords of a song, or visual representations of the audio such as its spectrum (Figure 5.2). A Vamp plugin may be non-causal and does not have to be able to run in real-time.

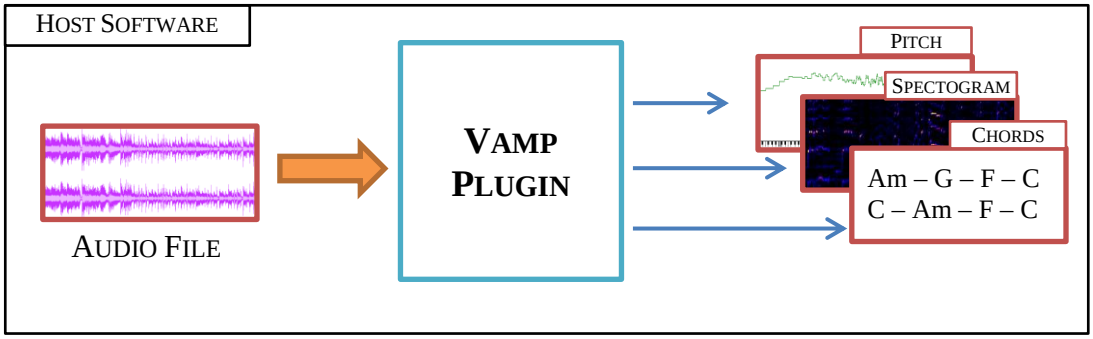

Figure 5.2 - Possible outputs of a Vamp Plugin when used in a host software like Sonic Visualiser.

A library of the existing Vamp plugins is available at [41].

5.2 The Qt library

The Sonic Visualiser's interface is developed using the *Qt library*. Qt is a cross-platform application framework that is used for developing application software with a graphical user interface (GUI) [42]. It was widely used by Nokia in the past to develop apps for smartphones that run the Symbian mobile operative system.

Qt Creator is the supporting Qt IDE (*Integrated Development Environment*) and the latest available version of Sonic Visualiser (2.1) requires Qt 5.0.2 to be compiled on Windows systems.

Qt is based on the concept of *widgets* that basically are objects in the GUI that can display data and status information, receive user input, and provide a container for other widgets that should be grouped together. A widget that is not embedded in a parent widget is called *window*.

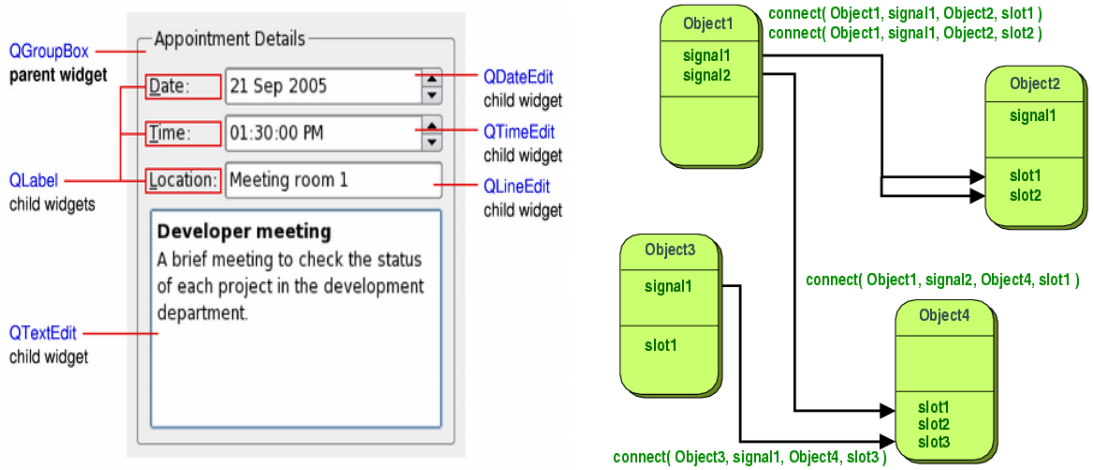

Figure 5.3 - The three Qt's fundamental elements: widgets, signals and slots.

The *signals* and *slots* mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. Signals and slots are used for communication between objects:

- A signal is emitted when a particular event occurs;
- a slot is a function that is called in response to a particular signal.

Qt's widgets have many predefined signals and slots, but it is possible to add custom ones to them.

5.3 Interface development

The integration of the IM AF Encoder in Sonic Visualiser is based on two main targets:

- Including a control for the tracks volume, in order to store this information in a preset in an IM AF file;
- Creating the "Export to IM AF file" option in the "File" menu of Sonic Visualiser, in order to save a multitrack session in an IM AF file.

The most part of the modification has been done on the *main window* code, editing *MainWindow.cpp* and *MainWindow.h* files inside the Sonic Visualiser's Qt project.

Sonic Visualiser's user interface is structured around *panes* and *layers*. A *pane* is a horizontally scrollable area of window, like a drawing canvas, that is opened when an audio file is imported into the session; a *layer* is one of a set of things that can be shown on a pane, such as the waveform associated to the audio file, the spectrogram or a frequency analysis. Every layer includes values about its task.

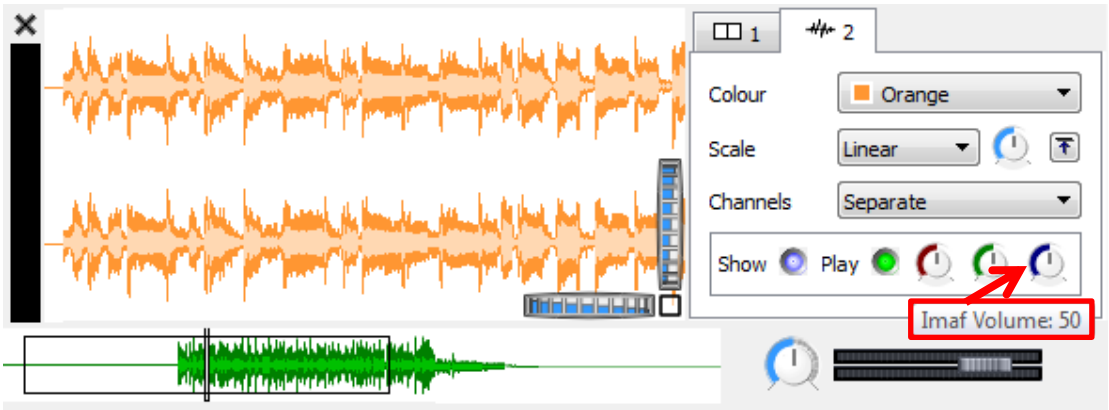

Figure 5.4 – Structure of a pane in Sonic Visualiser with the *AudioDial* control for the track volume in the IM AF file.

For the inclusion of the "*IM AF Volume*" control for each track in a Sonic Visualiser's session, we refers to the *Waveform* layer; the control is created by using the *AudioDial* widget placed next to the gain and pan controls of the layer, as shown in Figure 5.4.

To recover parameters from a layer, we need to scroll the widget's hierarchy of the main windows. This hierarchy is presented in Figure 5.5.

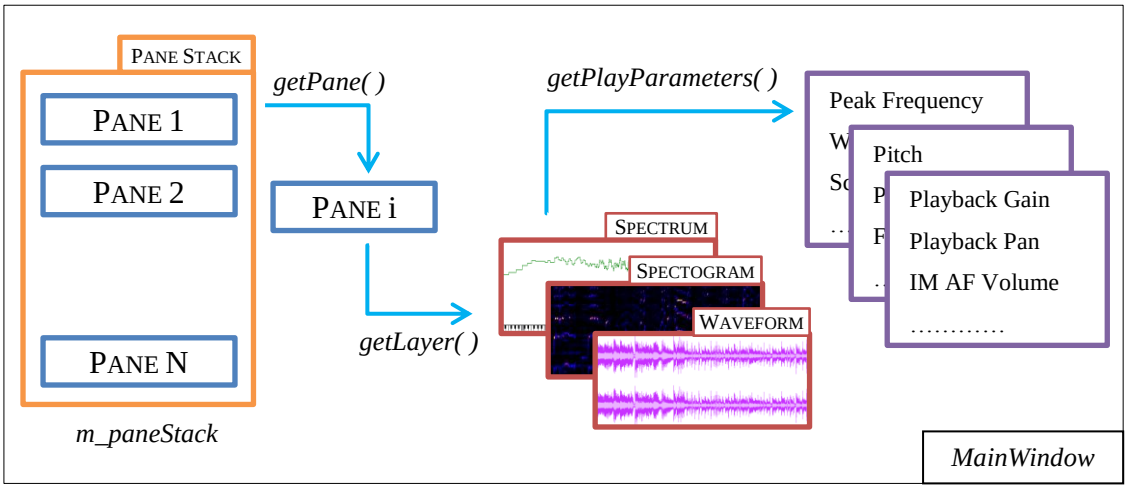

Figure 5.5 - Hierarchy of panes, layers and parameters in Sonic Visualiser, with the corrispective functions to recover the elements.

Once the parameters container for the waveform layer is recovered from the interested pane, the *IM AF Volume* parameters is obtained through the *getVolImaf()* function. The value varies between 0 and 100 and is passed to the *mainIMAFencoder()* function (that is the *main* of the developed encoder) when this is called to create the resulting IM AF file.

The second step of the integration is the creation of an option that allows exporting the Sonic Visualiser's multitrack session in IM AF format. A dialog window has been developed for this purpose and it appears when the "Export to IM AF file" option is selected from the "File" menu, as shown in Figure X.

The window allows selecting in a very easy way the features to include in the IM AF file that is going to be created. Some parameters are missing due to the choices made during the encoder implementation (they are set in the source code, see Table 4.1 - Paragraph 4.2) and to keep the dialog window more clean and user friendly.

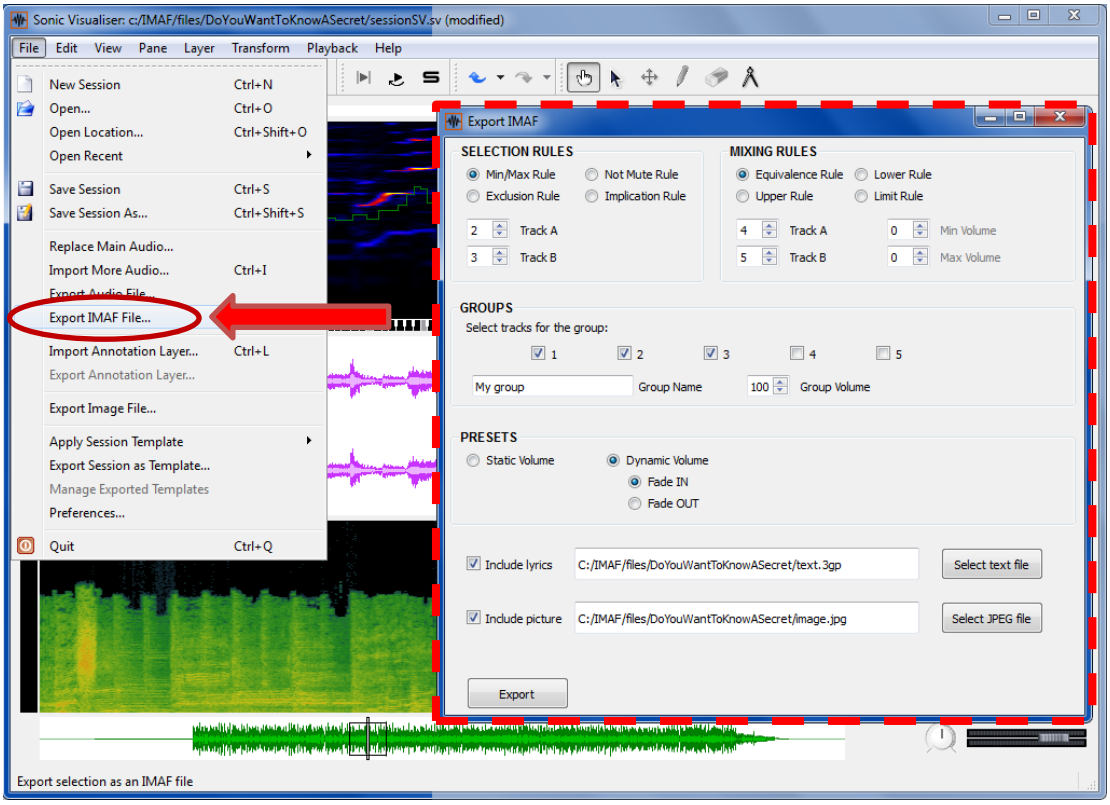

Figure 5.6 - Export option and encoder interface in Sonic Visualiser.

5.4 Scenarios

Providing IM AF support in a tool for the analysis of audio data, such as Sonic Visualiser, would open new opportunities to include media data in a file and to develop new tools for the music information retrieval (MIR).

A possible use case scenario is presented in the following using a Vamp plugin for chords extraction, known as *Chordino* [18]. In Figure 5.7 is shown the difference on using the same Chordino algorithm, with the same set-up, between the single instrument tracks and the classic mix-down of a song. Processing of the mixed track cannot be precise enough for having a reliable chords extraction, due to the overlap of several instruments. A more accurate extraction is achieved by applying the Chordino plugin to each single instrument track available in an IM AF file (i.e. on the guitar track only, for extracting the guitar chords). Moreover, it could be possible to perform automatic transcription of the chords in the timed-text line (aligned with the lyrics).

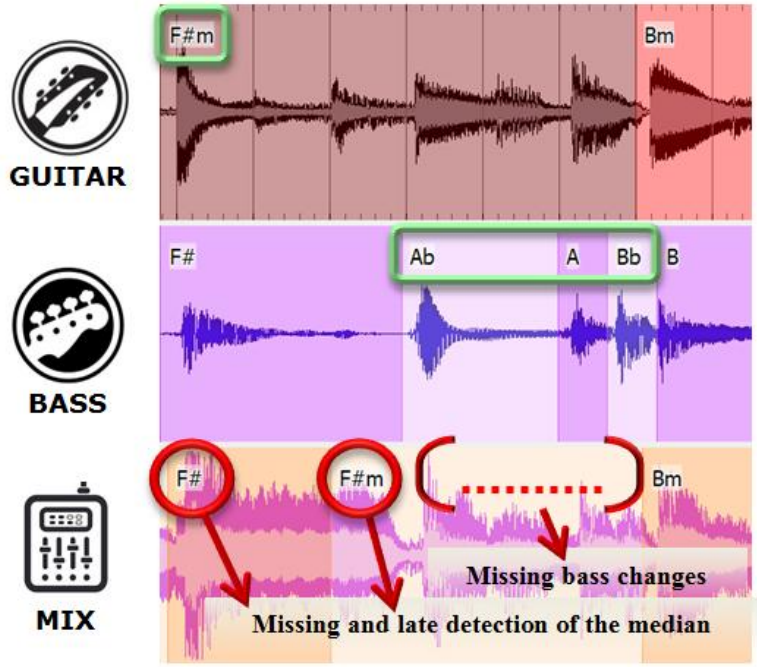

Figure 5.7 - Usage of Chordino in Sonic Visualiser for chords extraction: a) from individual music instruments (*guitar* and *bass*) and, b) the mix-down track (*mix*).

The same discussion, as the one about chords extraction, could be done for the extraction of the melody pitch of a song. Using the Vamp plugin called *Melodia* [21], an algorithm for melody pitch estimator, it is possible to have a more efficient pitch extraction from the only vocal track, rather than the mixed song. Extracting the lyrics from the vocal track (singing to text) could be another possible scenario, having an automatic way to elaborate information about the timing and the highlighting of the text for karaoke application without the necessity to create external 3GPP files, as it happens in the current implementation of the IM AF encoder. There is not yet an implementation for this kind of plugin; IM AF support in Sonic Visualiser could encourage such developments offering a test-bed for comparisons among different algorithms (singing voice to text, source separation for music instruments extraction from an MP3, etc).

Another use case scenario can be done using one more Vamp plugin called MATCH [10], an algorithm for audio alignment between two renditions of the same piece of music. IM AF could allow users to add their own musical instrument/vocal track in a song or replace the existing ones; alignment between the tracks is needed for having a proper resulting mix, ensuring that all the instruments start at the same point and follow the same beat. A tool like MATCH can carry out this task easily. Although Sonic Visualiser can run the MATCH plugin appropriately for calculating automatic alignments between two tracks, it does not contain any way for the user to provide multiple audio files as a set of individual channels for input to a Vamp plugin. This is a further demonstration of how important the IM AF integration in Sonic Visualiser is for the MIR community.

5.5 Future developments

IM AF integration in Sonic Visualiser opens new horizons for MIR researchers offering a test-bed for the development of a large number of projects and new VAMP plugins, among others, such as:

- on source separation for extracting individual music instruments/vocals from a mix-down song version;
- on singing voice to text conversion;
- on automatic highlighting of lyrics for karaoke applications.

In this way, some of the IM AF features could be automatically included in files, rather than to require producers to insert the chords in a text file along the lyrics at production time. For example, chords could be introduced in the interactive file using VAMP plugins like Chordino in Sonic Visualiser to automatically extract the chords of a song or from its individual music instruments. The same could be done for the lyrics from the vocal track.

About the IM AF Encoder, some proposals for further improvements can be:

- the possibility to apply more than one rule per time, using groups as well. Every element (track or group) is identified by its ID; an idea could be to provide a list of the available IDs, from which the user can pick the elements for a specific rule;
- metadata imported from the single MP3s (extracting ID3 tags) or inserted by user through the UI in Sonic Visualiser;
- exploring the possibility for pictures to perform a track, not just still pictures, for having a sort of "presentation";
- provide album support, using the file structure for a multiple type file (more than a *Movie Box* '*moov*' in the same file).

CONCLUSIONS

The creation of example IM AF files tested the full compliancy of the proposed encoder to the standard and it was useful to understand the benefits brought by this innovative file format. IM AF supports multitrack audio with volume sliders for DJ mixing and lyrics for karaoke applications, allowing storage of several instrument/vocal tracks in a single file alongside still pictures, lyrics and chords. All these features enrich the user's interaction space and make this format an interesting alternative for listeners to the use of other audio file formats, such as MP3 and Ogg.

There are also some psychological implications produced by the interactive player's interface. Many studies look at how specific visual features can enhance the listening experience [20].

It's proven that a transcript of the spoken word content as well as a waveform display or a particular picture related to the album, provide additional cognitive, emotional and aesthetic benefits to users. For example, nowadays that music business is mainly based on online selling and there's a loss of taste in younger listeners to buy physical supports, and since users may not have

access to some old fragile analog recordings, even an image related to the original recording media, or the particular time period of the performance for live albums, might increase interest towards the artist and its work.

Again, availability of lyrics as text synchronized with the song allows for the listener to follow along in a more enjoyable way, especially for compositions in a foreigner language. Moreover, presence of chords released officially by the artist will increase interest of users with playing skills to learn a song, avoiding useless research of frequently wrong tabs on internet.

Visual content in a nice interface can certainly enhance the user experience, as long as an option that allows to select or deselect features is available, for old-fashioned users who just want to listen.

By the presented use-case scenarios, we saw that using a multitrack file format like IM AF in a software like Sonic Visualiser could be advantageous for MIR researchers and can allow the automatic extraction/inclusion of the media data supported by the standard, such as timed text for chords or lyrics.

The file format has potentials that still have to be explored, and this could give to music companies the opportunity to revitalize the music market.

New on-line music forums and social networks could come to the world: personal mixes of songs could be exported and easily shared between users, due to the flexible features inherited from ISO-BMFF that allows to have lighter files containing only information about the mixing parameters, while the audio tracks can be made available through various on-line music services (e.g., groups, presets and rules are stored into the file; audio tracks, text and pictures are stored on a server linked by URLs).

This could consequently enable an efficient exchange and sharing of IM AF files in social networks. Each audio track could even be replaced by users' personal recordings, encouraging people to develop singing and music instruments playing skills through active learning.

A well-developed interactive music player with an ad-doc integration in the main internet browsers could support the launch of the mentioned web-services. It could be possible to parse/decode the file on the server side, in order to individually send to the client the components (audio, pictures, text, metadata). This allows the file belonging to brands with high number of audio tracks to be played even on devices with limited processing power.

IM AF has good potential for further exploration that enables it to be the format that will reign the digital audio world in the foreseeable future.

APPENDIX

Here is presented the *IM_AF_Encoder.h* header file code. It describes the structure of the boxes in an IM AF file.

```
#ifndef IM_AF_Encoder_IM_AF_Encoder_h
#define IM_AF_Encoder_IM_AF_Encoder_h

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

#define maxtracks 8
#define maxgroups 3
#define maxpreset 10
#define maxrules 10
#define maxfilters 3 //Max number of Filters for an EQ preset
#define maxdynamic 2 //Max number of Dynamic Volume changes
#define num_ch 2 //Number of channel outputs (STEREO)

#define files_path "/…/" //Put MP3, JPEG and 3GP files in this folder, home folder in "C:\"
#define output_file "example.ima" //Name of the output file
#define image_path "/…/image.jpg" //Direct link to the image

typedef long long u64;
typedef unsigned int u32;
typedef unsigned short u16;
typedef unsigned char u8;

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 MovieBox //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  // extends FullBox('hdlr')
         {
             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 //extends FullBox('smhd')
             {
                 u32 size;
                 u32 type;
                 u32 version;
                 u16 balance; // =0 place mono tracks in stereo. 0 is center
                 u16 reserved; // =0
             }SoundMediaHeaderBox;
                        struct NullMediaHeaderBox //extends FullBox('nmhd')
             {
                                    u32 size;
                 u32 type;
                 u32 flags;
                             }NullMediaHeaderBox;
```

```
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 TextSampleEntry{
                                          u32 size;
                                          u32 type; //tx3g
                                          u32 a;
                                          u32 b;
                                          u32 displayFlags;
                                          u8 horizontaljustification;
                                          u8 verticaljustification;
                                          u8 backgroundcolorrgba[4];
                                          u16 top;
                                          u16 left;
                                          u16 bottom;
                                          u16 right;
                                          //StyleRecord
                                          u16 startChar;
                                          u16 endChar;
                                          u16 fontID;
                                          u8 facestyleflags;
                                          u8 fontsize;
                                          u8 textcolorrgba[4];
                                          struct FontTableBoX{
                                                  u32 size;
                                                  u32 type;
                                                  u16 entrycount;
                                                  u16 fontID;
                                                  u8 fontnamelenght;
                                                  u8 font[5]; //Serif
                                          }FontTableBox;
                                  }TextSampleEntry;
         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 predifined;
                                 }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 == 0) || (preset_type == 8) - Static track volume preset
                struct StaticTrackVolume{
                        struct presVolumElem{
                                u8 preset_volume_element;
                                struct EQ{ // if preset_type == 8 (with EQ)
                                        u8 num_eq_filters;
                                        struct Filter{
                                                u8 filter_type;
                                                u16 filter_reference_frequency;
                                                u8 filter_gain;
                                                u8 filter_bandwidth;
                                        }Filter[maxfilters];
                                }EQ;
                        }presVolumElem[maxtracks];
                }StaticTrackVolume;

    // if (preset_type == 1) || (preset_type == 9) - Static object volume preset
                struct StaticObjectVolume{
                        struct InputCH{
                                u8 num_input_channel;
                        }InputCH[maxtracks];
                        u8 output_channel_type;
                        struct presElVol_1{
                                struct Input{
                                        struct Output{
                                                u8 preset_volume_element;
                                        }Output[num_ch];
                                        struct EQ_1{ // if preset_type == 9 (with EQ)
                                                u8 num_eq_filters;
                                                struct Filter_1{
                                                        u8 filter_type;
                                                        u16 filter_reference_frequency;
                                                        u8 filter_gain;
                                                        u8 filter_bandwidth;
                                                }Filter[maxfilters];
                                        }EQ;
                                }Input[num_ch];
                        }presElVol[maxtracks];
                }StaticObjectVolume;

    // if (preset_type == 2) || (preset_type == 10) - Dynamic track volume preset
                struct DynamicTrackVolume{
                        u16 num_updates;
                        struct DynamicChange{
                                u16 updated_sample_number;
                                struct presVolumElem_2{
                                        u8 preset_volume_element;
                                        struct EQ_2{ // if preset_type == 10 (with EQ)
                                                u8 num_eq_filters;
                                                struct Filter_2{
                                                        u8 filter_type;
                                                        u16 filter_reference_frequency;
                                                        u8 filter_gain;
                                                        u8 filter_bandwidth;
                                                }Filter[maxfilters];
                                        }EQ;
```

```
                                        }presVolumElem[maxtracks];
                                }DynamicChange[maxdynamic];
                        }DynamicTrackVolume;

                // if (preset_type == 3) || (preset_type == 11) - Dynamic object volume preset
                        struct DynamicObjectVolume{
                                u16 num_updates;
                                struct InputCH_3{
                                        u8 num_input_channel;
                                }InputCH[maxtracks];
                                u8 output_channel_type;
                                struct DynamicChange_3{
                                        u16 updated_sample_number;
                                        struct presElVol{
                                                struct Input_3{
                                                        struct Output_3{
                                                                u8 preset_volume_element;
                                                        }Output[num_ch];
                                                        struct EQ_3{ //if preset_type==11 (with EQ)
                                                                u8 num_eq_filters;
                                                                struct Filter_3{
                                                                   u8 filter_type;
                                                                   u16 filter_reference_frequency;
                                                                   u8 filter_gain;
                                                                   u8 filter_bandwidth;
                                                                }Filter[maxfilters];
                                                        }EQ;
                                                }Input[num_ch];
                                        }presElVol[maxtracks];
                                }DynamicChange[maxdynamic];
                        }DynamicObjectVolume;

        // if (preset_type == 4) || (preset_type == 12) - Dynamic track approximated volume preset
                        struct DynamicTrackApproxVolume{
                                u16 num_updates;
                                struct DynamicChange_4{
                                        u16 start_sample_number;
                                        u16 duration_update;
                                        struct presElVol_4{
                                                u8 end_preset_volume_element;
                                                struct EQ_4{ // if preset_type == 12 (with EQ)
                                                        u8 num_eq_filters;
                                                        struct Filter_4{
                                                        u8 filter_type;
                                                u16 filter_reference_frequency;
                                                u8 end_filter_gain;
                                                                u8 filter_bandwidth;
                                                }Filter[maxfilters];
                                                }EQ;
                                        }presElVol[maxtracks];
                                }DynamicChange[maxdynamic];
                        }DynamicTrackApproxVolume;

// if (preset_type == 5) || (preset_type == 13) - Dynamic object approximated volume preset
// THIS STRUCTURE GIVES STACK OVERFLOW PROBLEMS - MORE STACK SIZE NEEDED -> Needs investigation
                        struct DynamicObjectApproxVolume{
                                u16 num_updates;
                                struct InputCH_5{
                                        u8 num_input_channel;
                                }InputCH[maxtracks];
                                u8 output_channel_type;
                                struct DynamicChange_5{
                                        u16 start_sample_number;
                                        u16 duration_update;
                                        struct presElVol_5{
                                                struct Input_5{
                                                        struct Output_5{
                                                                u8 preset_volume_element;
                                                        }Output[num_ch];
                                                        struct EQ_5{ // if preset_type == 11 (with
EQ)
```

```
                                                                u8 num_eq_filters;
                                                                struct Filter_5{
                                                                   u8 filter_type;
                                                                   u16 filter_reference_frequency;
                                                                   u8 end_filter_gain;
                                                                   u8 filter_bandwidth;
                                                                }Filter[maxfilters];
                                                        }EQ;
                                                }Input[num_ch];
                                        }presElVol[maxtracks];
                                }DynamicChange[maxdynamic];
                        }DynamicObjectApproxVolume;

        char preset_name[50];

    }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;
                    // Only for Min/Max Rule
                    // if (selection_rule_type==0)
                            u16 min_num_elements;
                            u16 max_num_elements;
                    // Only for Exclusion and Implication Rules
                    // if (selection_rule_type==1 || selection_rule_type==3)
                            u32 key_element_ID;
                    char rule_description[20];
    }SelectionRules;
    struct MixingRules{
        u32 size;
        u32 type;
        u32 version;
        u16 mixing_rule_ID;
        unsigned char mixing_type;
        u32 element_ID;
                    u16 min_volume;
                    u16 max_volume;
        u32 key_elem_ID;
        char mix_description[17];
    }MixingRules;
}RulesContainer;
struct GroupContainerBox{ //extends Box('grco')
    u32 size; // = 10 + sizeGRUP
    u32 type;
             u16 num_groups;
             struct GroupBox{ // extends FullBox('grup')
             u32 size; // = 21 + 15 + 30 (+4 if group_activation_mode = 2)
             u32 type;
             u32 version;
             u32 group_ID;
             u16 num_elements;
             struct groupElemId{
                     u32 element_ID;
    }groupElemId[maxtracks];
             unsigned char group_activation_mode;
             u16 group_activation_elements_number;
             u16 group_reference_volume;
             char group_name[22];
             char group_description[32];
```

```
                }GroupBox[maxgroups];
        }GroupContainerBox;
}MovieBox;

typedef struct MetaBox // extends FullBox ('meta')
{
        u32 size;
        u32 type;
        u32 version;
        struct theHandler //extends FullBox HandlerBox('hdlr')
        {
                u32 size;
                u32 type;
                u32 version; // version = 0 + flag
                u32 pre_defined; // =0
                u32 handler_type; // = 'meta' for Timed Metadata track
                u32 reserved[3]; // =0
                unsigned char name[4];
        }theHandler;
        struct file_locations //extends Box DataInformationBox('dinf')
        {
                u32 size;
                u32 type;
/*              struct DataReferenceBox2
                {
                        u32 size;
                        u32 type;
                        u32 flags;
                        u32 entry_count;   // = 1
                        struct DataEntryUrlBox2 //extends FullBox('url', version=0, flags)
                        {
                                u32 size;
                                u32 type;
                                u32 flags;
                        }DataEntryUrlBox;
                }DataReferenceBox; */
        }file_locations;
        struct item_locations //extends FullBox ItemLocationBox('iloc')
        {
                u32 size;
                u32 type;
                u32 version; // version = 0 + flags
                unsigned char offset_size; // = 4 bytes
                unsigned char lenght_size; // = 4 bytes
                unsigned char base_offset_size; // = 4 bytes
                unsigned char reserved; // = 0
                u16 item_count;  // = 1
                u16 item_ID;  // = 1
                u16 data_reference_index;  // = 0 (this file)
                u32 base_offset; // size=(base_offset_size*8)=4*8
                u16 extent_count;   // = 1
                u32 extent_offset; // size=(offset_size*8)=4*8
                u32 extent_length;  // size=(lenght_size*8)=4*8
        }item_locations;
        struct item_infos //extends FullBox ItemInfoBox('iinf')
        {
                u32 size;
                u32 type;
                u32 version; // version = 0 + flag
                u16 entry_count;   // = 1
                struct info_entry// extends FullBox ItemInfoEntry('infe')
                {
                        u32 size;
                        u32 type;
                        u32 version; // = 0
                        u16 item_ID; // = 1
                        u16 item_protection_index; // = 0 for "unprotected"
                        char item_name[6]; // name with max 5 characters
                        char content_type[18]; // = 'application/other' -> 17 characters
                        char content_encoding[4]; // = 'jpg' for JPEG image -> 3 characters
                }info_entry;
```

```
        }item_infos;
        struct XMLBox // extends FullBox('xml ')
        {
                u32 size;
                u32 type;
                u32 version;
                char string[2000];
        }XMLBox;
}MetaBox;

typedef struct MediaDataBox // extends Box('mdat')
{
    u32 size;
    u32 type;
    unsigned char data;
}MediaDataBox;


#endif
```

REFERENCES

[1] ISO/IEC 23000-12:2010 - Information Technology - Multimedia Application Format (MPEG-A) -- Part 12: *Interactive Music Application Format*.

[2] ISO/IEC 23000-12:2010/Amd.1:2011 - Information technology - Multimedia application format (MPEG-A) - Part 12: *Interactive music application format*, AMENDMENT 1: *Conformance and reference software.*

[3] ISO/IEC 23000-12:2010/Amd.2:2012 - Information technology - Multimedia application format (MPEG-A) - Part 12: *Interactive music application format*, AMENDMENT 2: *Compact representation of dynamic volume change and audio equalization*.

[4] ISO/IEC 14496-12:2008 - Information technology - Coding of Audio-Visual Objects – Part 12: *ISO base media file format*.

[5] ISO/IEC 10918-1:1994 - Information technology - *Digital compression and coding of continuous-tone still images (JPEG).*

[6] ETS 3GPP TS 26.245-2004 - Transparent end-to-end Packet switched Streaming Service (PSS); *Timed text format*.

[7] ISO/IEC 15938-5:2003 - Information technology - Multimedia content description interface – Part 5: *Multimedia description schemes*.

[8] ISO/IEC 23003-2:2010 - Information technology - MPEG audio technologies - Part 2: *Spatial Audio Object Coding (SAOC).*

[9] C. Cannam, C. Landone and M. Sandler: "Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files", *Proceedings of the ACM Multimedia 2010 International Conference*.

[10] S. Dixon and G. Widmer: "MATCH: a music alignment tool chest", *Proceedings of the International Symposium on Music Information Retrieval,* pp. 492-497, 2005.

[11] C. Grigg: "Preview: Interactive XMF - A Standardized Interchange File Format for Advanced Interactive Audio Content", *115th Audio Engineering Society Convention*, October 2003

[12] J. Han, Z. Rafii and B. Pardo, “Audio Source Separation” and “REPEAT”, Research projects of Northwestern University, Dep. of Elec. Eng. and Comp. Sc., http://music.cs.northwestern.edu

[13] T. Hosoya, M. Suzuki, A. Ito and S. Makino: “Lyrics recognition from a singing voice based on finite state automation for music information retrieval”, *Proceedings of the International Symposium on Music Information Retrieval*, pp. 532-535, 2005.

[14] iKlax Media, http://www.iklaxmusic.com

[15] 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.

[16] P. Kudumakis, "MP3: something's gotta change!", Audio!, pp. 6, Vol. 1, Issue 3, April 2011. Editors: P. Curzon, M. Barthet and S. Dixon.

[17] L. A. Ludovico: “Key concepts of the IEEE 1599 Standard”, *Proceedings of the IEEE CS Conference The Use of Symbols To Represent Music And Multimedia Objects*, pp. 15-26, 2008.

[18] M. Mauch and S. Dixon: “Approximate Note Transcription for the Improved Identification of Difficult Chords”, *Proceedings of the International Symposium on Music Information Retrieval*, pp. 135-140, 2010.

[19] MOGG files, Multitrack Digital Audio Format, http://moggfiles.wordpress.com

[20] A. Murray and J. Wiercinski, “Looking at archival sound: Enhancing the listening experience in a spoken word archive”, article on FirstMonday.org, Volume 17, Number 4 - April 2nd 2012.

[21] J. Salamon and E. Gómez: “Melody Extraction from Polyphonic Music Signals using Pitch Contour Characteristics”, IEEE Transactions on Audio, Speech and Language Processing, 20(6):1759-1770, Aug. 2012.

[22] Mac Developer Libray – “QuickTime 6.3 + 3GPP”, http://developer.apple.com

[23] Smule, Inc. – “Glee Karaoke”, “I am T-Pain” – www.smule.com

[24] Activision Publishing, Inc. – “Guitar Hero” – www.guitarhero.com

[25] Harmonix Music Systems, Inc. – “Rock Band” – www.harmonixmusic.com

[26] E. Onate, "Development an IM AF encoder" – MSc Digital Music Processing, Queen Mary University of London, 2012. Available at: https://code.soundsoftware.ac.uk/hg/enc-imaf

[27] Audizen, http://www.audizen.com (last viewed, February 2011)

[28] MT9 file format – http://en.wikipedia.org/wiki/MT9

[29] ChoJin-seo, "New MP3 Revolutionizes Way You Listen to Music" - Korea Times - http://www.koreatimes.co.kr

[30] Audacity – http://audacity.sourceforge.net

[31] Song Galaxy – http://www.songgalaxy.com/

[32] MXP4 – http://en.wikipedia.org/wiki/MXP4

[33] Bopler – https://www.facebook.com/BoplerGames

[34] MOD format – http://en.wikipedia.org/wiki/MOD_(file_format)

[35] ISO-BMFF, ISO base media file format http://en.wikipedia.org/wiki/ISO_base_media_file_format

[36] MiraVid MP4 Browser, available online at http://download.cnet.com/MiraVid-MP4-Browser

[37] CodePlex MP4 Explorer, available online at http://mp4explorer.codeplex.com

[38] Sound Software - http://soundsoftware.ac.uk/

[39] Mercurial CVS - http://mercurial.selenic.com/

[40] Easy Mercurial - http://soundsoftware.ac.uk/easymercurial

[41] Vamp Plugins - http://www.vamp-plugins.org/

[42] Qt Project - http://qt-project.org/

[43] MP3 file structure - http://www.multiweb.cz/twoinches/mp3inside.htm