H3rtz.stdio

Final Report

Jean "Areas" Bou Raad
Kevin-Brian "KB" N'Diaye
Thanh Lam "Vellelah" Nguyen
Yabsira "Yabs" Alemayehu MULAT

Contents

Chapter 1

Introduction

In this last report, we will present the achievements of our team over the entire semester. Every reported issue has been fixed and the main goals for this project have been achieved.

We will dissect each part of our project step by step, look back on the progress we have made and share our thoughts on the project. The first part of our report will be about the pulse-audio backend and the WAV file format.

The second one will bring details on the implemented first and second phase of the MP3 encoding.

The third chapter introduces the new file format based on Huffman coding.

Finally, the final look of the user interface will be shown. The part showcases the integration between the algorithmic part and the UI and its evolution.

Chapter 2

Wave file format

In this part, we will be providing explanation on our work to encode and decode the wave file format.

2.1 Introduction to the format

The wave file format is an old audio file format developed by Microsoft and IBM released in 1991. This standard is nowadays wildly adopted by major operating systems such as Windows, Linux distributions and many more.

This vast adoption was possible thanks to the use of a standard to encode the data: the **R**essource **I**nterchange **F**ile **F**ormat also known as the *RIFF*.

Wave files can contain various types of audio data: uncompressed and compressed format are possible. Usually, we find uncompressed audio data under the form of **P**ulse-**c**ode **m**odulation (PCM). However, we can also find MPEG compressed data and many other kinds of compression.

Over the years, the WAV format evolved thanks to various updates made by Microsoft. Today, in addition to audio data, one can find various metadata in the files concerning the track played. Also, the encoding process has slightly evolved.

2.2 IO: handling files

Before going into the details of our decode, we need first to introduce systems implemented to read files.

We use the system calls to open first a file and then map it in memory using the function *mmap(2)*. The program puts the data in this structure:

```c
typedef struct file
{
    int fd; // file descriptor
    unsigned char *map; // data
    struct stat *stats; // statistics concerning length and more...
} file;
```

Figure 2.1: The file structure

Of course, we need methods to unload the file. It can be done using the *munmap(2)*. It requires the size as parameter which justifies the presence of the stat structure.

2.3 Decoding process

In this sub-part, we will be explaining the process that allows us to retrieve information from a wave file and to play its audio. Jean Bou Raad with the help of Kevin-Brian N'Diaye created this part.

2.3.1 Introduction to headers

Headers constitute the central piece of the puzzle to understand the data in a file. They provide essential information such as the size of each sample (8-bits, 16-bits, 32-bits, 64-bits), the sampling frequency [1] (typical values are 44100, 48000, 96000 Hz), and more.

Using a hexadecimal file viewer, we can see that an audio file looks like this:

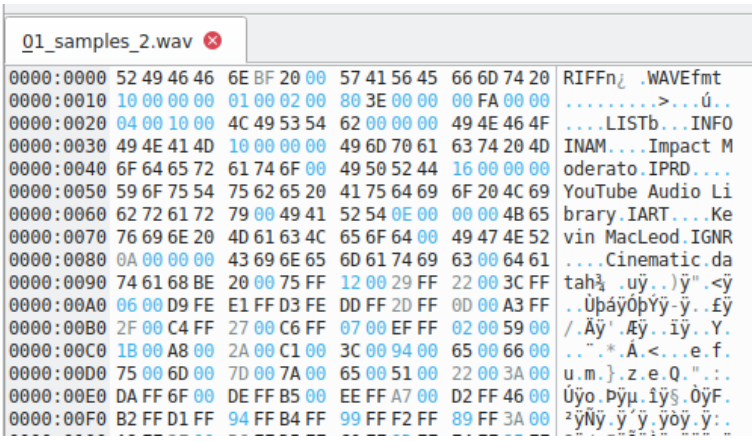


Figure 2.2: A typical wave file

[1] number of audio points taken when recording the audio

For any human being, it is mostly unreadable data. But, one can find some patterns with the strings "RIFF", "WAVE", "DATA", "INFO" appearing. Those point the existence of metadata out. The standard defines those blocks of data as chunks. There are a limited set of blocks with their specific definition:

1. RIFF: a short block which confirms the type of the file. It usually comes out as the first block in a wave file.
2. FMT: short name for "format". It is a mandatory chunk which provides the various characteristics of the audio. There are multiple variations of this chunk.
3. LIST: Linked list chunk containing information on the file.
4. FACT: Short chunk which contains no useful information for our audio player.
5. DATA: it contains the audio data (samples). The audio is either compressed or uncompressed.

Various blocks and formats mean various implementations, challenges that we have successfully overcome.

2.3.2 Parser: main execution loop

The objective of the parser is to go through the whole file to find each block of data and assign it to structures that the program can manipulate.

Hence, the main process features a loop that iterates until it reaches the end of the file. At each step, it will check if an identifier is here and do the appropriate calls to sub-functions.

For each kind of block, a parsing method is available:

1. RIFF: *riffParser*
2. FMT: *fmtParser*
3. LIST: *listParser*
4. FACT: *factParser*
5. DATA: *dataParser*

They take as parameter the pointer to the index variable to move it according to the block's size. They also allocate memory for each structure which are explained in the later parts. The main structure refers to each of these substructures and are as follow:

```c
// this generic type contains all wav chunks
// Warning: some can be null
typedef struct wav
{
    riff *riff; // cannot be modified
    union // store both types at the same place: memory efficient
    {
        fmt *fmt;
        fmt_float *fmt_float;
        fmt_extensible *fmt_extensible;
    };
    fact *fact; // optional: often used for float
    struct data *data; // unvariable
    fmt_type format; // casting helper
    list *list;
} wav;
```

Figure 2.3: Definition of the main wave header structure

Reminder: in the C programming language, the union is used to refer to a variable with multiple types. We use it here to gather three variations of the *fmt* block. These elements are all pointers so they share a common size which makes the union possible.

2.3.3 Parser: the RIFF chunk

The first chunk to parse is one of the easiest: the RIFF. As said in the introduction, it stands for **R**essource **I**nterchange **F**ile **F**ormat. It contains an identifier which allows the parser to check that the file is really a WAV file[2].
Focusing on the sample file from before, we can isolate the RIFF chunk to study it:

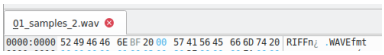


Figure 2.4: RIFF chunk

It contains two visible strings: "RIFF" and "WAVE". In the middle, it contains the total size of the file minus the first 8 bytes. In the hierarchy of blocks, this one sits on top of any other kind of block.

To reproduce that structure in the program, we defined the following structure:

[2]Otherwise it aborts the process

```
typedef struct riff
{
    char riff[4];//contains 4 bytes
    int fileSize; //contains 4 bytes
    char fileFormatId[4]; // contains 4 bytes
} riff;

```

Figure 2.5: Riff structure definition

Riff and *fileFormatId* contain most of the time the strings "RIFF" and "WAVE" are without their null terminating byte. The process is really trivial as you can see:

```
// Riff chunk parser
// @param data: the mapped data in memory
// @param i: index in the file
riff *riffParser(unsigned char *data, size_t *i)
{
    riff *riff = malloc(sizeof(struct riff));
    strcpyn(data + *i, riff->riff, 4);
    *i += 4;
    riff->fileSize = intConversionLE(data + *i);
    *i += 4;
    strcpyn(data + *i, riff->fileFormatId, 4);
    *i += 4;
    return riff;
}

```

Figure 2.6: Riff parsing function

As the data stream is made of bytes, it is not easy to retrieve the value of multiple bytes. intConversionLE is a function that converts four bytes into an standard integer. The process behind this function is explained in the next part.

2.3.4 Tool: merging bytes

As mentioned above, most of the characteristics strings are not null terminated. This means that we cannot use unprotected functions from the standard libraries (*strlen* to only give one example). We are doing many comparisons between strings during the parsing process so an equivalent of *strcmp* was needed. Its protected counterpart *strncmp* could have done the job just fine for most of the tasks. However, we also need to support both little-endian and big-endian representations of data. For instance, all strings are in big-endian representation in the wave file format.

Hexadecimal value	String
0x52494646	"RIFF"
0x57415645	"WAVE"
0x666d7420	"FMT"
0x64617461	"DATA"

Figure 2.7: Example of strings with their hexadecimal representation

Hence, we decided to go for the implementation of a byte merger with various sizes: short, integer, long, and 128-bits integers. You can find below an example of byte merging tool:

```
1 // Fast 4 bytes to int conversion (merging) BIG ENDIAN
2 // @param data: must be an at least 4 bytes long array
3 __uint32_t intConversionBE(unsigned char *data)
4 {
5     return (data[0] << 24) | (data[1] << 16) | (data[2] << 8) | (
      data[3]);
6 }
7
```

Figure 2.8: Conversion function from bytes to 32-bits integers

This solution is fast and safe. Also, it allowed the implementation of many facilitating tools with enumerations and switches. For instance, to identify the numerous information identifiers, we used an enumeration such that each value is at the index of the identifier's big-endian representation:

```
typedef enum infoIds
{
    IARL = 0x4941524C, // archival location
    IART = 0x49415254, // artist
    ICMS = 0x49434D53, // commisionned
    ICMT = 0x49434D54, // comments
    ICOP = 0x49434F50, // copyright
    ICRD = 0x49435244, // creation date
    // skipped some codes (image related)
    IGNR = 0x49474E52, // genre
    IKEY = 0x494B4559, // keywords
    INAM = 0x494E414D, // name
    ISBJ = 0x4953424A, // content of the file
    ISFT = 0x49534654, // software
    ISRC = 0x49535243, // source
    IPRD = 0x49505244, // Product
    IPRT = 0x49505254, // track id
} infoIds;

```

Figure 2.9: Enumeration of information identifiers

These give some easier-to-implement switches in our code to handle the numerous cases:

```
    // converts a 4-bytes string into an integer value
        switch (uintConversionBE((unsigned char *)current->infoId))
        {
        case IARL:
            f->archival = current->data;
            break;
        case IART:
            f->artists = current->data;
            break;
        // etc... more cases
        case IPRD:
            f->product = current->data;
            break;
        case IPRT:
            sscanf(current->data, "%d", &f->tracknb);
            break;
        case ISBJ:
            f->album = current->data;
            break;
        default:
            warnx("unknown attribute %s %s", current->infoId,
    current->data);
            break;
        }

```

Figure 2.10: Example of simplified code thanks to the use of enumerations

If we had not implemented this solution, it would have been a lot of if and else if statements with many strings comparisons. For this specific case, we reduced the 13 series of comparisons to a long but fast switch statement.

However, some identifiers are 128-bits wide and cause some issues. GCC, our compiler, does not fully support 128-bits integers. For instance, this declaration will cause an error:

```
static const __uint128_t PCM_CODE2 =
0x00000001000001080000aa00389B71;
```

While this one will not:

```
const __uint128_t PCM_FLOAT_CODE =
((__uint128_t) 0x0300000000001000 << 64) | 0x800000aa00389b71;
```

GCC cannot directly evaluate literal 128-bits expression. Instead, we hack our way in by generating an expression that respects GCC's 64-bits limitation.

2.3.5 Parser: the FMT chunk

This part of the header is one of the most important part. FMT stands for FORMAT. It can be extended to the notion of audio format which answers these questions: how many bits per samples, are they compressed, the bitrate, and more... It is crucial for playback because PulseAudio needs this information to properly reproduce the sound correctly.

Basic format chunk

This part comes with some difficulties as there are **three** variations of that part of the header. However, they all share common characteristics with this basis:

```
typedef struct fmt
{
    char fmt[4]; //format block id 4B
    int blocSize; // classic 16b

    unsigned short AudioFormat; //2B see enum compression_codes
    short channel; //2B

    int sampling_freq; // 4B
    int bitrate; //important 4B //number of bytes to read per sec
    short blocrate; //2B NbrChannels * BitsPerSample/8
    short samplerate; //2B //8-16-24 bits
} fmt;

```

Figure 2.11: The first FMT structure (V1)

This basic part of the header contains various attributes:

1. fmt: the "FMT " string without the null terminating byte
2. blocSize: this block's size, for the basic FMT block it is 16 bytes.
3. AudioFormat: the wave file format is a container format which can hold various kinds of compressions. Below you can find an enumeration of the most used compression codes:

```
//most common compression modes for wav
enum compression_codes
{
    any = 0, //if the format doesn't need this info
    PCM = 1, //used
    ADPCM = 2,
    PCM_float = 3, //might be used for 32-64bits formats
    alaw = 6,
    Amu_law = 7,
    IMA_ADPCM = 17,
    Yamaha = 20,
    GSM = 49,
    G721 = 64,
    MPEG = 80, // used
    WaveFormatExtensible = 65534, //used
    Experimental = 65536,
};

```

Figure 2.12: Compression codes for the Wave format

4. Channel: the audio can be mono (1), stereo (2), or multi-channel.

5. Sampling frequency: the number of samples taken per second (Hz). Typical values are 44100 Hz, 48000Hz, 96000Hz.

6. BitRate: the number of bits required to have one second of audio.

7. BlocRate: the number of bytes per block with the number of channels taken into account in the computation.

8. sampleRate: the bit-depth, represents the quality of each sample. This value is usually 8-bits, 16-bits, 32-bits, 64-bits with more being better quality. From 32-bits, it is often PCM float data and not regular integer data.

PCM-Float format chunk

If the file has a float-PCM data, the FMT contains more data. The structure is as follows:

```
// 18 bytes fmt block for IEEE float PCM
typedef struct fmt_float
{
    char fmt[4];
    int blocSize; // will be 18

    unsigned short AudioFormat;
    short channel;

    int sampling_freq;
    int bitrate;
    short blocrate;
    short samplerate;
    // all of the above is the same
    short wExtSize; // new attribute
} fmt_float;

```

Figure 2.13: FMT float variation (V2)

WExtSize is an extra attribute that is not used anywhere in the project.

WaveExtensible format chunk

In one of its revisions, Microsoft introduced an extended header with new metadata. This extra metadata allows more flexibility and adapts the format to the relatively new technologies[3]. This variation is more interesting because it contains interesting metadata.

To recognize this variation, we can check the blocSize attribute (40 bytes instead of 18) or check the compression code which will be 65534 (see figure 2.12).

The variation is as follows:

[3] We are talking about Windows 2000 (February 2000), new...

```
// 40 bytes bloc WAVE_EXTENSIBLE
typedef struct fmt_extensible
{
    char fmt[4];
    int blocSize; // will be 40

    unsigned short AudioFormat;
    short channel;

    int sampling_freq;
    int bitrate;
    short blocrate;
    short samplerate;
    // above is nothing new
    short extension_size; //sub-block size
    short valid_bytes_ps;
    int channel_mask; // new indicator for surround sound with
    multi-channels support
    char sub_format[16]; //contains the new format identifier
} fmt_extensible;

```

Figure 2.14: Wave Extended format chunk (V3)

This extension uses the compression code to highlight its existence. Hence, without further parsing, the program is unable to recognize which format. The compression code is in another place: sub_format which is 16 bytes long (128-bits long). We implemented only a few compression codes because 128-bits integers are complicated to handle in C99 as mentioned in part 1.2.4.

Hexadecimal value	FORMAT
0x01000000000001000800000aa00389b71	PCM
0x03000000000001000800000aa00389b71	PCM FLOAT

Figure 2.15: Compression code

Handling variations

To handle these variations in the code easily, we use union as explained at figure 2.3. The first version of the block is common to the other variations. Hence, this part is shared in memory, which means that whatever the variation, the basic header is filled with at least this information. Then, using a casting helper, we can access more data if it is available. This solution is memory efficient, fast, and easy to implement.

2.3.6 Parser: the FACT chunk

This chunk is short and contains minor information. It is as follows:

```
typedef struct fact
{
    char fact[4]; //"FACT"
    int chunk_size; // 4B
    int nb_samples;
} fact;

```

Figure 2.16: Typical FACT structure declaration

2.3.7 Parser: the DATA chunk

This chunk is essential because it contains audio samples.

```
typedef struct data
{
    char data[4]; // "DATA"
    int data_bytes; //number of bytes in the data chunk
    unsigned char *chunk; // entry point for real data
} data;

```

Figure 2.17: Typical DATA structure declaration

Using the bitrate fromm the FMT chunk and the attribute data_bytes, the program can guess the total duration of the audio file.

2.3.8 Parser: the LIST chunk

The list is specific in terms of structure. It indicates the beginning of a list of elements inside the current block. It has no predefined size. Its main purpose is to hold information about the music itself. For instance, it can contain the title, the album, the date of publication, the software used to encode the file, and more... This block is optional.

The variable nature of this block led to a special structure. We adopted a linked list with a sentinel to parse the data.

```
1  // Contains various info
2  // linked list with sentinel structure
3  typedef struct info
4  {
5      char infoId[4];
6      unsigned int size;
7      char *data;
8      struct info *next;
9  } info;
10
11 typedef struct list
12 {
13     char list[4];
14     int chunk_size;
15     unsigned char *data;
16     struct info *infos;
17 } list;
18
```

Figure 2.18: Definition of the INFO and LIST chunks

Each node contains an identifier. It says what the data is about (see figure 2.9).
We implemented methods to allocate and free the linked list.

2.4 Encoding process

This is part was done by Kevin-Brian N'Diaye, with Thanh Lam Nguyen's participation on the GUI linking. This part will mostly be based on the parser used by the WAV encoding part.

What we needed to do for the first part was being able to take a WAV file and recreate the same file using our parser.

As this part is modular, it will flow directly into the next part: Adding more information given by the user.

First of all, we split the WAV file into two:

1. The .raw file containing the data section of the parser.
2. The new file containing the new header.

Keeping the raw signal will be really useful for the conversion support. However, important information such as:

1. Audio type (type of signal)
2. Channels
3. Estimated duration

Without those information, the signal won't be read correctly no matter what the format. In the future, a structure allowing an easy flow of those information from a format to another will be very useful for conversion support.

Encoding an intact WAV file was quite straight forward but changing it had its challenges.

Thus, we will encode the data based on the WAV_EXTENSIBLE convention which looks like this using Okteta.

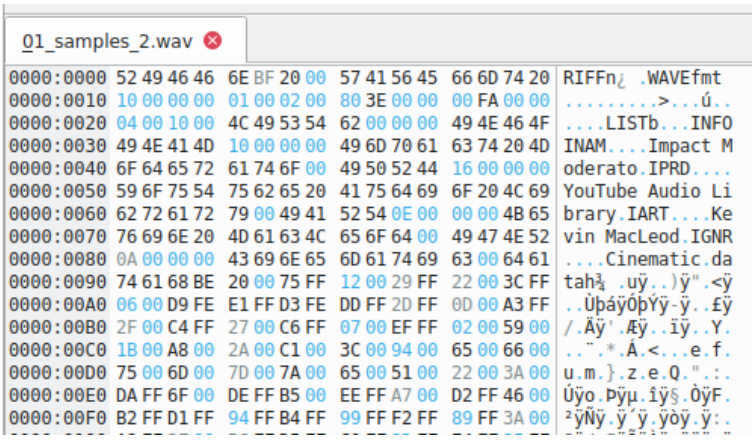


Figure 2.19: WAV_EXTENSIBLE header format

The WAV_EXTENSIBLE stems from the XMP metadata convention where the storage of metadata in files follows a few rules:

1. The first part of a chunk is its tag then its size
2. 3 null bytes will follow the size of the current chunk
3. If the size of the chunk is odd, a null bytes is added to balance everything out. (It's called padding)
4. And the rest contains conventions for what to name each chunk (artist, genre and etc...)

2.4.1 Writing the new header

Using the terminal and now the GUI, we can add new information to the file.

Things like:

1. The archive location
2. The artist
3. Copyrights
4. Creation date
5. Genre
6. Title
7. etc..

Basically, any new information gets added to the linked list called list. It contains information about the music itself which can be lacking for certain files.

The function write_header uses the fd from the new_file called old_file_2.wav and a new header (either from the old file or the user).

The main function in the encoding process is called:

```
1 void write_wav(char *raw_file, char **new_wav, struct wav *header);
```

We can now expand around this function with other function either supplying the necessary parts or using it to implement it in the GUI.

```
1 void write_raw(char *path, char **raw_file, struct wav *header);
```

This function simply creates a new raw file with the PCM signal.

Because PCM signal are uncompressed, we are dealing with a large amount of bytes. Therefore, two functions were written to avoid errors or worse, signal corruption.

```c
// safe write
void rewrite(int fd, const void *buf, size_t count, char *err_msg)
{
    int r;
    size_t offset = 0;
    while (offset < count)
    {
        r = write(fd, buf + offset, count - offset);
        if (r == -1)
        {
            r = errno;
            errx(EXIT_FAILURE, "failed to write %s into fd: %s",
    err_msg, strerror(r));
        }
        if (r == 0)
        {
            break;
        }
        offset += r;
    }
}

// safe read
void reread(int fd, void *buf, size_t count, char *err_msg)
{
    int r;
    size_t offset = 0;
    while (offset < count)
    {
        r = read(fd, buf + offset, count - offset);
        if (r == -1)
        {
            r = errno;
            errx(EXIT_FAILURE, "failed to read %s from fd: %s",
    err_msg, strerror(r));
        }
        if (r == 0)
        {
            break;
        }
        offset += r;
    }
}
```

They work based on the server programming practical but some changes were made to improve error handling. The *err_msg* parameter is there to help us spot which part of the encoding process failed and the *errno* integer gives us additional information on the error.

```
//writes n null bytes for spacing
void write_space(int fd, size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        unsigned char hex = 0x0;
        if (write(fd, &hex, 1) < 0)
        {
            errx(1, "encode: write spacing failed");
        }
    }
}
```

2.4.2 GUI version

The linking is part of the second and final version of the WAV encoding.
When the only thing the codec does is copy-paste, there's no point in using it which renders our work useless.
Additionally, if it's useless to the user it doesn't have its place in the GUI and thus, the final part.

Therefore, we needed a way to link the WAV codec to the GUI.

Adding information

The main reason someone might want to tinker with audio files to begin with is to add new information which is what we will do with the WAV codec.

The user will be able to change and/or add information from the file if they choose to do so.

The information is "limited" to $header \rightarrow list \rightarrow infos$ linked list as it contains most of the metadata.

```
for (; current != NULL; current = current->next)
{
    int art = strncmp(current->infoId, "IART", 4);

    if (art == 0 && strcmp(argv[0], "No changes\n") != 0)
    {
            unsigned int diff = strlen(argv[0]) - strlen(current->
    data);
            current->size += diff;
            header->list->chunk_size += diff;
            header->riff->fileSize += diff;
            current->data = argv[0];
            check[0] = 1;
            continue;
    }
}
```

This template mostly explains how we can update the information from the info list.

The parameters are the list of arguments called argv which the info given by the user. It follows a convention we created which goes as follow:

0 Artist

1 Copyrights

2 Genre

3 Name

4 Album

If no information is given by the user, no information is updated. If the element already exists (artists, name, etc...), it's just being replaced after some updates to every relevant size parameter. The check list needs to know whether the string was used.
Otherwise, we have to the create the element ourselves like so:

```
1 strcpyn((unsigned char *)"IART", (char *)artist->infoId, 4);
2 artist->size = strlen(argv[0]);
3 artist->data = argv[0];
4 artist->next = NULL;
5 header->riff->fileSize += artist->size;
6 header->list->chunk_size += artist->size;
7 current->next = artist;
8 current = current->next;
```

Mostly the same thing but here we have to use the linked list structure carefully as to not lose any data by simply replacing a node instead of adding one to the list. The relevant size parameters are still being updated and we can write all that new information to a new file.

Linking

How do we link those functions to the GUI then? We create a button to show different fields where the user can change them, press enter and have those changes written into a new file.

Those fields will not be empty if the information already exists. Here's what it looks like for the final version:

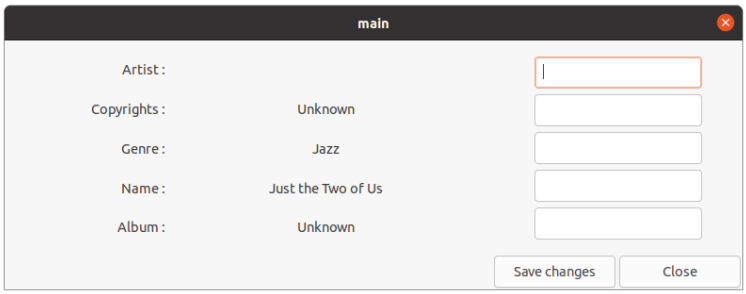


Figure 2.20: New window to input new information

2.4.3 Software used

Because a broken header results in broken audio, we had to use a new software to read the raw data itself to debug our progress.

We used Okteta, a raw data editor. It allowed to see where and how the data is assembled in the header. But most importantly, it allowed us to debug our functions for the WAV encoding.

For example, we noticed that 3 null bytes would be written after every chunk size.

2.5 Conclusion

The WAV codec has been a great introduction to audio codecs. Some metadata convention was present and resources were sufficient. It allowed us to use great tools like Okteta, multithreading and low-level I/O on a middle size project. We were not able to transfer those experiences to a full MP3 codec which was our biggest regret. In the end, learning about codecs has been a great experience for us as it taught about low-level I/O.

Chapter 3

PulseAudio - Multi-threaded API

A major part of our project is about being able to provide an audio-playback experience. In the C programming language, there are multiple tools able to do that[1]. They vary from very low level to high level when it comes to communicating between the program and the audio drivers. As a compromise, we chose PulseAudio for that.

PulseAudio is a software provided on many Linux distributions. It handles the audio playback between software and drivers via a server. It also provides an C API, which enables communication between programs and the audio server running.

The API provides both synchronous and asynchronous methods with a multi-threaded of the last one. Our project integrates a GUI which needs to run while playing back the audio. Hence, we needed an asynchronous implementation with multi-threading. However, PulseAudio's API/library is thread aware but not thread-safe[2]. It means that the implementation needs more care to ensure its stability.
In this part, we will go in the details of this complicated process with various examples.
The main developer on this part is Jean Bou Raad.

3.1 Definitions

This part is essential, we explain essential types and we will not explain them in later parts!

[1] GStreamer, Alsa, and more...
[2] https://www.wikiwand.com/en/Thread_safety

Before going into the code implementation of the audio player, some definitions need to be provided.

In PulseAudio's library, one can find various objects with their role. Below, you can find a short definition of each types:

1. *pa_threaded_mainloop* (mainloop): A thread based event loop implementation based on pa_mainloop. The event loop is run in a helper thread in the background. A few synchronization primitives are available to access the objects attached to the event loop safely. It can act as a mutex for object access. This type plays a major role in thread safety.

2. *pa_stream* (stream): An opaque type for playback. Objects of this type communicate audio samples to the server. It also provides useful information during playback (latency, volume, and more...)

3. *pa_context* (context): An opaque type used to retrieve information about an audio server.

3.2 Main steps of implementation & features

This part of the project has for sole purpose playing back audio. This is not a trivial process. Here are the major steps:

1. Creating a mainloop.

2. Connecting to the default audio server.

3. Creating a stream that links a pipe to a server.

4. Draining the stream or interrupting it.

In addition to these main steps of implementations, we implemented some additional features:

1. Volume: we implemented methods that change the volume and gets the current one from the PulseAudio server.

2. Latency: we implemented a function that retrieves the current latency from the software to the audio server. It is essential for accurate timestamp computation.

3. Multiple formats support: PulseAudio can only take PCM audio in (raw signal). Compressed formats are not supported by the library[3].

We will review each steps and features in the parts below.

[3]except when using pass-through (the device can decode the compressed signal)

3.3 Storing objects in structures

PulseAudio's library comes with a variety of objects and types. They have various uses and can vary from one context to another. We mentioned in the second report a rework that focused on that matter because the organization was bad.

Our final structures separate objects in two categories: variable and static. Static objects are the ones that we need to initialize once in the program's lifetime. The variable objects are the ones that can change between tracks for instance.

The following structures are the ones mainly used:

```
1 typedef struct pa_player
2 {
3     wav_player *player; //variable objects
4     pa_objects *pulseAudio; //static objects
5     pa_info *info; // for volume
6     state pa_state; // global PulseAudio State
7     fileType type;
8 } pa_player;
9
```

Figure 3.1: PulseAudio player structure definition

```
1 // Per track data, will change over the program's lifetime
2 typedef struct wav_player
3 {
4     wav *info; // contain header and data pointers for the file
      played
5     file *track; // I/O attributes
6     unsigned char *data; // points to the beginning of the audio
      samples
7     pa_stream *stream; // Opaque PulseAudio object to launch
      operations
8     drain *drainer; // holds information about the draining status
9     pa_time *timing; // position in the current file, latency
10    playerStatus status; // see enum
11 } wav_player;
12
```

Figure 3.2: Variable objects: wav_player structure

```
// Static PulseAudio objects during the program's lifetime
typedef struct pa_objects
{
    pa_context *context;
    pa_threaded_mainloop *loop;
    pa_mainloop_api *api;
    char *sink;
    pa_server_info *server;
} pa_objects;

```

Figure 3.3: Static objects: PulseAudio objects structure

As you can see, the objects are separated clearly between static and variable elements. Everything is dynamically allocated. We have methods that do these memory allocations with standard error handling.

3.4 Implementation

3.4.1 Multi-threading, callbacks, signals: how it works

As mentioned above, PulseAudio's library relies on deferred callback[4] to give data about its state. They are essential when using multi-threading if we want to convert asynchronous operations into synchronous ones. Let's see an example of that principle in action:

```
void getDefaultSink(pa_player *player)
{
    pa_objects *pa = player->pulseAudio;
    pa_threaded_mainloop_lock(pa->loop);
    pa_operation *op = pa_context_get_server_info(pa->context, &
    callbackSink,
                                                  player);
    while ((state = pa_operation_get_state(op)) !=
    PA_OPERATION_DONE)
    {
        pa_threaded_mainloop_wait(pa->loop);
    }
    pa_operation_unref(op);
    pa_threaded_mainloop_unlock(pa->loop);
}

```

Figure 3.4: A typical asynchronous callback function converted into synchronous code

[4] see https://www.wikiwand.com/en/Callback_(computer_programming)

This code is as generic as it gets. First, we lock the mainloop just like a mutex. This avoids race conditions by preventing the access to PulseAudio's objects. Then, we launch our asynchronous operation. It can be any function that returns an operation object. However, here it is the pa_context_get_server_info function. The prototypes vary from one operation to another but we always find as parameter a function (here the second parameter). This is the function called when the operation is done (it can fail). This callback has a very important function, it can send a signal which unlocks the loop at line 9. Once, the operation is done, we can unreference the object and unlock our mainloop.

```
1  void callbackSink(pa_context *c, const pa_server_info *i, void *
       userdata)
2  {
3      assert(c);
4      pa_player *player = userdata;
5      pa_objects *pa = player->pulseAudio;
6      int error = asprintf(&(pa->sink), "%s", i->default_sink_name);
7      if (error <= 0)
8          err(errno, "Couldn't copy sink name");
9      // this signal unlocks the waiting function
10     pa_threaded_mainloop_signal(pa->loop, 0);
11 }
12
```

Figure 3.5: Typical deferred callback associated to figure 3.4

This is the simplest example. We will see in later parts that sometimes we must handle operations differently. However, parts of the skeleton you have seen remains.

3.4.2 States

Using multi-threading implies many safety issues, especially when running parallel tasks. The last part explained how we prevent race conditions. However, we did not explain how we avoid invalid operations. This is crucial because it avoids many crashes. For instance, trying to kill a stream which does not exist leads to a segmentation fault. The solution is quite trivial: when the mainloop is locked, the player contains a few attributes that provide information on its state.

First, the structure pa_player has the pa_state variable of type state. State is an enumeration with these caracteristics:

```
typedef enum state
{
    BABY, // before mainloop init
    READY, // before stream init
    ACTIVE, // while playing/paused
    TERMINATED, // killed stream but track in memory <=> ready
    FINAL, // killed mainloop
} state;

```

Figure 3.6: The state enumeration for the PulseAudio player structure

The states READY and TERMINATED are equivalent, meaning that we can go from one to another. Only the FINAL state is definitive. These states refer mainly to PulseAudio's objects' states.

The ACTIVE state means that a wav_player is currently playing audio. To know if it is playing, paused, or not ready, we implemented an enumeration. It is a subset of the ACTIVE state.

```
typedef enum playerStatus
{
    NOT_READY, // if state != ACTIVE
    PLAYING, // can be drained
    PAUSED, // corked stream
} playerStatus;

```

Figure 3.7: The player status enumeration

We will display the player's states during each operation with a table. If the original state do not match the player's ones then the program will abort the operation.

3.4.3 Creating a PulseAudio player using the library

As we said before, our implementation requires multi-threading. PulseAudio provides a thread-aware mainloop in its API which acts as mutex in addition to its event polling role. To initialize our PulseAudio player, we follow those steps:

1. Create a threaded mainloop with pa_threaded_mainloop_new.
2. Get the API associated to the new mainloop via pa_threaded_mainloop_get_api
3. Create a new context using the new API with pa_context_new
4. Start the threaded mainloop
5. Connect the context to the device (sink) with pa_context_connect

6. Wait for the context to be ready using deferred callbacks

7. When the state is PA_CONTEXT_READY, we can return. Our player is almost to play files!

The player states are:

Entity	Initial State	New State
pa_player	BABY	READY
wav_player	NOT_ACTIVE	NOT_ACTIVE

3.4.4 Creating a stream from a Wave file

We need to initialize the playback on PulseAudio to create a stream. The function in charge of doing that process is as follows:

```
1 int prepareStream(pa_player *player);
2
```

Figure 3.8: The prepare stream function

To create a classic stream, one needs:

1. An already parsed wave file: the *wav_player* must hold a file with its data and characteristics. The steps below depend on this.

2. Sample specifications: they come from the file that we need to read. For this, we created a function that matches the characteristics of a wave file to its *pa_sample_spec*.

3. A channel mapping: the type *pa_channel_map* represents the channel characteristics of a file. Briefly, it tells PulseAudio if the stream will be mono or stereo.

4. Buffer attributes: it gives PulseAudio information on how one wants to handle the buffer. For our project, we tell PulseAudio to process it automatically. It can be for audio streaming software (network streams).

5. Flags: they tell PulseAudio how to handle the stream in various ways. Most of them are used to tell the library to handle the things by itself (latency, timings, for instance). Those features are headed towards server developers.

6. Finally, by using *pa_stream_new_extended* and *pa_stream_connect_playback*, we can create a stream and connect it to the device (sink).

This process is long. And with multi-threading in mind, we need to integrate state-checking. Its goal is preventing any invalid operation on the player. Here, that can be creating a new stream while one is still playing, or trying to create

```
1 void callback_write(pa_stream *stream, size_t requested_bytes, void
      *userdata);
```

Figure 3.9: Declaration of PulseAudio write callback function

a player without the required objects initialized.

The player states are as follows:

Entity	Initial State	New State
pa_player	READY or TERMINATED	ACTIVE
wav_player	NOT_ACTIVE	PAUSED[5]

3.4.5 Sending audio samples to play sound

This part is critical because it must be fast and reliable. The callback function is as follows: The library provides the two first parameters. We can choose freely the last one. The process is as follows in a while loop till we have written on the buffer the *requested_bytes*:

1. Determine the number of bytes we want to write: this can be arbitrary.
2. Initialize a buffer with *pa_stream_begin_write*. It handles memory allocation automatically.
3. Fill the buffer with data from the file. It can be accessed through the *pa_player structure*, via the *wav_player*, and the *wav* object inside. The block DATA contains a pointer to it. For simplicity, the reference is copied directly as an attribute of *wav_player* (see fig. 3.2).
4. Write it to the real buffer with *pa_stream_write*.
5. Update the offset for the file and the number of bytes written.

We do not need to check for states in this operation. PulseAudio calls it automatically when needed (acts as a deferred callback).

3.4.6 Pausing & Resuming audio

In the essential set of features, we have the playing and pausing functionality. PulseAudio provides methods to do these operations. The method used for both of them is pa_stream_cork. It is an asynchronous callback that we transformed into a synchronous ones to change states. The updated states are as follows:

Entity	Initial State	New State
pa_player	ACTIVE	ACTIVE (no change)
wav_player	PAUSED	PLAYING[6]

If the initial states requirements are not met by the player, the program will give warnings and abort the operation.

```
1     // we first lock the mutex before accessing variables
2     pa_threaded_mainloop_lock(pa->loop);
3     if (player->pa_state != ACTIVE)
4     {
5         warnx("Pause: no stream playing, can't stop playback...");
6         pa_threaded_mainloop_unlock(pa->loop);
7         return;
8     }
9     if (player->player->status != PLAYING)
10    {
11        warnx("Pause: already paused...");
12        pa_threaded_mainloop_unlock(pa->loop);
13        return;
14    }
```

Figure 3.10: State checking in the pausing function

3.4.7 Terminating & Draining a stream

Terminating a stream is an important operation. It must be handled correctly because it can create critical bugs.

The operation itself of terminating a stream is fairly straightforward. We just created a function that runs an asynchronous operation with the function *pa_context_suspend_sink_by_name*, then we can disconnect the stream with *pa_stream_disconnect*.

The problematic is not how but when. Indeed, PulseAudio cannot determine when a stream is done. So, to prevent a situation where we have audio samples left to play, we must drain the audio buffer before killing the stream.

PulseAudio provides built-in function to drain a buffer. It is again an asynchronous operation. However, it is more complicated than previous operations. Indeed, the draining operation fails after a certain amount of time if we still feed the buffer. In the most basic situations, it does not happen. However, if we rewind in the track (see section on offsetting) while that operation is running, the draining fails. To solve this particular issue, we implemented a structure which contains the draining operation and the draining states. They are as follows:

```c
typedef enum DrainStatus
{
    DRAIN_INACTIVE, // no operation running
    DRAIN_ACTIVE, // operation running -> playing state
    DRAIN_FINISHED, // can interrupt stream
} DrainStatus;

// Object used to track draining process
typedef struct drain
{
    DrainStatus state; //enumeration
    pa_operation *drain; // PulseAudio operation
} drain;
```

Figure 3.11: Structure and enumeration linked to 3.2

With these structures introduced, we can now provide the expected states for each object before the draining operation:

Entity	Initial State	New State
pa_player	ACTIVE	ACTIVE
wav_player	PLAYING	PLAYING
drain	DRAIN_INACTIVE	DRAIN_FINISHED

During the operation, the drain has a state at DRAIN_ACTIVE. As usual, if the player does not meet those conditions, it will abort the operation and return safely.

In addition to this implementation of states, we added a way to cancel the operation if needed. It is something unique in our code in terms of structure. Indeed, the drain function wrapper works as a synchronous function to update states correctly. Hence, we need to use another thread to cancel the operation and then handle correctly the cancellation on the main thread. The cancel function is as follows:

```c
void cancelDrain(pa_player *player)
{
    wav_player *pa = player->player;
    // state checking removed for simplicity
    pa_operation_cancel(pa->drainer->drain);
    pa_threaded_mainloop_signal(player->pulseAudio->loop, 0);
}
```

Figure 3.12: The cancelDrain function (asynchronous operation)

This function can only work with those particular states:

Entity	Expected
pa_player	ACTIVE
wav_player	PLAYING
drain	DRAIN_ACTIVE

As you can see in the snippet of code, we cancel the operation, then send a signal to the mainloop. It unlocks the main thread which will check the operation's state. If it is indeed cancelled, it restores the player to its previous state:

```
1     pa_operation_state_t state;
2     while ((state = pa_operation_get_state(op)) !=
      PA_OPERATION_DONE)
3     {
4         // if the operation is cancelled by another thread
5         if (state == PA_OPERATION_CANCELLED)
6         {
7             warnx("drainStream: cancelled operation...");
8             // we restore the callbacks
9             pa_stream_set_write_callback(player->player->stream, &
      callback_write, player);
10            // cleanup the memory
11            pa_operation_unref(op);
12            player->player->drainer->drain = 0;
13            player->player->drainer->state = DRAIN_INACTIVE;
14            // unlock our mutex
15            pa_threaded_mainloop_unlock(pa->loop);
16            // return safely
17            return;
18        }
19        // before the signal from cancelDrain
20        pa_threaded_mainloop_wait(loop);
21    }
```

Figure 3.13: Extract of the drain function (see 3.4 for reference)

What remains is the stream termination. This process is not difficult and involves no particular notions. For reference, see *pa_context_suspend_sink_by_name* in PulseAudio's documentation.

3.4.8 Offsetting: back and forth in the tracks

Offsetting is one of the most complicated operations on PulseAudio. It is user-input based and can happen at any time during playback.

There are multiple ways to approach the problem. For instance, we could modify our way of writing by writing on the buffer's reading offset. This is complicated, so, we went for the easiest choice: we flush the buffer and write new samples according the new offset. PulseAudio provides an asynchronous operation to do that: pa_stream_flush. We need to update the reading offset then. The operation's core is as follows:

```
1     pa_operation *op = pa_stream_flush(player->player->stream, &
      cb_seeking, player);
2     while (pa_operation_get_state(op) != PA_OPERATION_DONE)
3         pa_threaded_mainloop_wait(pa->loop);
4     pa_threaded_mainloop_unlock(pa->loop);
5
6     // offset computation with block alignment
7     player->player->timing->offset =
8                     offset - offset % player->player->info->fmt->
      samplerate;
9
```

Figure 3.14: Snippet of the relative function

Line 7 is particularly interesting as it provides a little insights on the way audio we read audio. We must align it to the locks size. Otherwise, it degrades the audio quality or can corrupt it completely.

Before rewinding the audio, we must make sure that a draining operation is not running to avoid errors. Therefore, we check the states and cancel the operation if needed using the function described in figure 3.12. The expected states are:

Entity	Initial State	Temporary State	New State
pa_player	ACTIVE	-	ACTIVE
wav_player	PLAYING	PAUSED	PLAYING
drain	INACTIVE or ACTIVE	DRAIN_INACTIVE	INACTIVE

If the drain's state is active, we cancel the draining operation.

To conclude, thanks to the precautions we took, this operation is stable and reliable. The states' rework for second defense is handy when it comes to the operation.

3.4.9 Volume

PulseAudio comes with volume management built-in. The documentation advises explicitly not to modify system volumes with the library. The main reason is that volume scales differently from device to device.
Hence, we implemented a way to modify the volume of the input of our stream.
PulseAudio implemented the volume as a double structure. Indeed, for multi-channel audio, PulseAudio can set a specific volume per channel (for instance, one for the left ear and another for the right). Each channel (represented by the structure *pa_cvolume*) has a *pa_volume_t* attribute which is the real volume of the current channel object. They are stored in an array in the *pa_cvolume* structure. The function to modify the volume works as follow:

```
void getVolume(pa_player *player);
void setVolume(pa_player *player);
```

Figure 3.15: two prototypes used to interact with the volume via PulseAudio

Information about the current volume of our player is stored in our structure *pa_info*:

```
typedef struct pa_info
{
    // modify that value to change the volume
    // must be a double between 0.0 and 1.0
    double volume;
    // sink input id
    uint32_t id;
} pa_info;
```

Figure 3.16: Declaration of the PulseAudio information structure

The attribute id is an unique identifier used to retrieve the volume attached to the stream. As mentioned before, this is not a system global variable. The volume is modified by the mainly modified by the GUI.

3.4.10 Timestamps

Like any regular audio player, we need to know the current timestamp while playing a file. It might look like a simple process, but It can be tricky. Indeed, the way to get the current timestamp of a played file is to divide the offset by the bitrate (number of bits per second of audio). However, this method is inaccurate. Indeed, PulseAudio is fed with audio data ahead of time. That means that the offset is always farther in the buffer than the audio currently played.
And so, getting latency is key. As usual, PulseAudio did not provide a guide to do it. Hence, there were many difficulties. We ended up using a simple function from the library *pa_stream_get_latency*. It retrieves *pa_usec_t* data, which is a long integer representing the number of microseconds of latency. With some more computations, we were able to retrieve the correct timestamp:

```
// current reading timming as a double, for instance we read the
  2.0s sample
double timing = pulseAudio->player->timing->time;
// converts to microseconds (multiplies by 10**6)
timing *= power;
// removes latency
double wLatency = timing - (double) *(pulseAudio->player->timing
  ->latency);
// current is a completion percentage (0-100%, 0 is the beginning
   and 100 the end)
double current = wLatency/power;
```

Figure 3.17: Accurate timestamp computation

3.4.11 MP3 and other formats playback

Due to high complexity of decompression, especially with the number of algorithms involved, we decided to avoid doing decompression. This led to the implementation of a fast audio decoder for compressed formats.

To do this, we are using Gstreamer, a library from the Gobjects family (like GTK) to decode audio. Indeed, gstreamer is specialized in media handling with numerous plugins and capacities. We use these plugins:

1. filesrc: retrieves data from a file
2. decodebin: decodes the audio data
3. audioconvert: converts the audio data (PCM)
4. wavenc: encodes the raw data to wave style format file.
5. giostreamsink: fake sink to retrieve data from it

This library shares similarities with PulseAudio and GTK in its functioning.
One important note, usually, the output of a file conversion is another file. However, we do not want to leave trash files. So, we used a fake sink to retrieve data and then play it.
The data retrieved is a WAV file. So, it can be combined by the program with the existing functions to play the audio.
The process can be slow. However, this was the best solution to build a complete audio player.

Warning: some of the required modules are not on EPITA's computer by default. Playback will fail in that case.

3.5 Conclusion

To conclude, PulseAudio is a great library. It offers many features and great freedom to developers by giving them choices in ways to implement features. However, it is sometimes a trap: documentation & examples lack on the internet.

Through the implementation of the multi-threaded music player, we consolidated many skills around multi-threading. It is the result of long struggles and can eventually lead to weird crashes due to race conditions[7].

Moreover, this part of the project is vast. We began with the most basic features (pure playback) and ended up with more complex ones (offsetting, volume). Implementing features on top of an already existing system can be somewhat difficult. However, if the code is built with maintenance and legibility in mind, then it becomes easier.

We mentioned GStreamer as our reference for other file format decoding, but in fact this part of the project is more or less a down-scaled version that library. Just as us, they use PulseAudio for playback. This fact confirms that PulseAudio is a standard in the industry and justifies our choice.

Finally, we are happy with this back-end. It offers the features we expected in the book of specifications with good stability.

[7] It killed one of our Manjaro virtual machines

Chapter 4

MP3 Encoding

4.1 Time-Frequency Filterbank

Applying filter to signals has the main advantage of getting rid of the useless aspects of the signal and reduce the size of the signal.

The MP3 standard recommends a high-pass filter to improve the sound quality while removing lower frequencies.

This is the first step of the first phase, where low frequencies are simply being cut out from the signal by default.

4.2 Analysis Subband Filter

This filter is a polyphase filter. It turns a PCM signal (from a WAV file) with fs as default sampling frequency into 32 equally spaced subsection by sampling frequencies of fs/32.

This polyphase filter is later completed by the MDCT, those two create a hybrid filterbank.

4.2.1 Implementation

We have to follow the following steps:

1. Divide the audio into 32 samples
2. Create a list X of 512 elements where the first 32 elements are the audio samples then:

$$X_i = X_{i-32}, \text{ for } i = 511 \text{ down to } 32. \tag{4.1}$$

3. Multiply the each coefficient by an constant array C to create an array Z

4. Create an array Y following this equation:

$$Y_i = \sum_{j=0}^{7} Z_i + 64j, \text{ for } j = 0 \text{ down to } 63. \tag{4.2}$$

5. Create the 32 subband samples S by matrixing with:

$$S_i = \sum_{k=0}^{63} M_{i,k} * Y_k, \text{ for } i = 0 \text{ to } 31. \tag{4.3}$$

6. The final equation which will give us the coefficients of the final matrix by the following formula:

$$M_{i,k} = \cos\left[\frac{(2i+1)(k-16)\pi}{64}\right], \text{ for } i = 0 \text{ to } 31. \tag{4.4}$$

The constant array C draws this function:

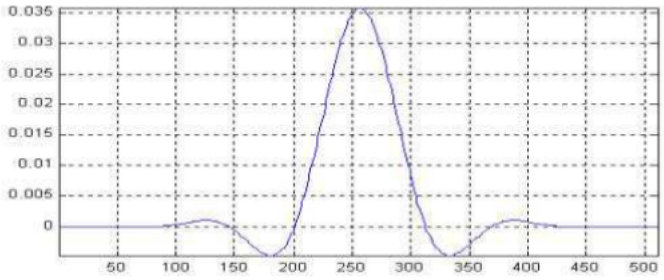


Figure 4.1: Coefficients from C_i

4.3 Psycho-acoustic model

The main developer for this part of the project is Jean Bou Raad with Kevin-Brian N'Diaye assistance for mathematical notions.

4.3.1 Introduction

The MPEG standard encoding process is lossy. It means that the original audio, in addition to being compressed, is also being modified to reduce its size. The process discards also some data. To determine which data to keep and delete, the standard uses a psycho-acoustic model.

The term psycho-acoustic itself is a part of science that studies the relation between the perception of sound in the human ears and the sound being sent. Here, in our encoding process, a psycho-acoustic model tries to imitate the perception of the sound and select only the sound that the human can hear. For instance, generally speaking, we cannot perceive sounds above 20 kHz.Hence, the sounds above that range will be discarded by the model. But, there's more to it than that. The implementation of model 1 for the MPEG 3 - Layer 3 audio encoding process requires vast algorithms that this part describes.

4.3.2 Definitions

This part of the project is not only about computer science. Hence, we decided to provide some definitions for some keywords.

1. Sound Pressure Level (SPL): represents the relative loudness of a sound. It can be negative or positive. Its unit is the decibel (dB).

2. Masking: Our ears cannot distinguish two sounds that are too close. The sound with the higher sound pressure will mask the other. The model uses them to discard some data.

3. Critical Bandwidth (critical band): the critical band is the band of audio frequencies within which a second tone will interfere with the perception of the first tone by auditory masking[1]. It is the first block to detect the masking effect.

4. Maskers: represents the pressures required to hear a sound at a frequency *f*. In our implementation, it is an array that scans ranges of frequencies. Unit is again dB.

[1] https://www.wikiwand.com/en/Critical_band

4.3.3 Steps in implementation

This small subsection details the global process in which audio samples go through to get a mask. It will be a quick overview. The following parts will explain the notions in depth.

1. Using a Fast Fourier Transform (FFT) algorithm, we transform a time signal into frequencies with relative sound pressure levels. The result is called the SPL array, and the indices are called spectral lines.
2. Determination of the sound pressure level in each subband. This process is not yet complete because it interacts with future parts of the project.
3. Finding tonal and non-tonal components in our spectral lines.
4. Discarding data that cannot be heard by the human ear. They are either too weak or too close to another tonal component.
5. Calculation of the maskers using the relevant data. We compute two of them: one for tonal components and one for noises.
6. Computation of the global masker.
7. Determination of the minimum masking threshold in each subband.
8. Computation of the subband masking ration (SMR).

4.3.4 FFT: getting the sound pressures

As explained above, the first step is about transforming our audio samples from a unit of time into a range of frequencies. First, we must normalize our input:

$$x(n) = \frac{s(n)}{N * 2^{b-1}}$$

Where $s(n)$ is the input array, N the number of samples in our array, and b the number of bits per sample.

Then, with $s(n)$, we compute the FFT, which is as follows:

$$SPL(k) = PN + 10 \log_{(10)} | \sum_{n=0}^{N-1} x(n)h(n) \exp{(-i\frac{2\pi kn}{N})}|^2$$

Where PN is the power normalization term, for MP3, it is required to scale the values with a maximum of 96 dB, $h(n)$ is the Hann window[2].

[2]Please refer to https://www.wikiwand.com/en/Hann_function

In the C language, it translates to this piece of code:

```
1   // memory allocation here, Hann window is computed also above
2   long double max = -INFINITY; //macro from standard
3   for (size_t i = 0; i < HALF; i++)
4   {
5       // FFT transform on NB_SAMPLES points => 1024 for MP3
6       // sum allows to retrieve total value faster
7       long double complex sum = 0;
8       for (size_t j = 0; j < NB_SAMPLES; j++)
9       {
10          long double complex val = cexpl(-I* ((2*M_PI*i*j)/
    NB_SAMPLES));
11          long double complex q = window[j]*samples[j]*val;
12          sum += q;
13      }
14      // conversion to real norm
15      long double norm = cabsl(sum);
16      long double sq = powl(norm, 2);
17      long double res;
18      // null logarithm is undefined!
19      if (sq != 0)
20          res = 10*log10l(sq);
21      else
22          res = sq;
23      vector[i] = res;
24      if (res > max)
25          max = res;
26  }
27  // normalization to 96DB max
28  long double PN = 96.0-max;
29  addToArrayl(vector, HALF, PN);
```

Figure 4.2: FFT transform code for MP3 encoding

As you can see, we determine the PN using the result of the left-hand side term. The vector is 512 points long, which is half of the 1024 points used. In reality, there is a symmetry between the part before the middle and after the middle of the array.

These points represent the relative pressure with a maximum level of 96 dB. Moreover, the points scan a range of frequencies, which are $\frac{fs}{N}$ Hz. For instance, if $fs = 44100$ Hz, then each point scans a range of approximately 43.066 Hz. Meaning that at $SPL(112)$ gives the relative pressure of the sounds in the range of frequencies of 4823.44 Hz. These points are also called spectral lines.

Using a Jupyter notebook and some python magic[3], we get the following graph:

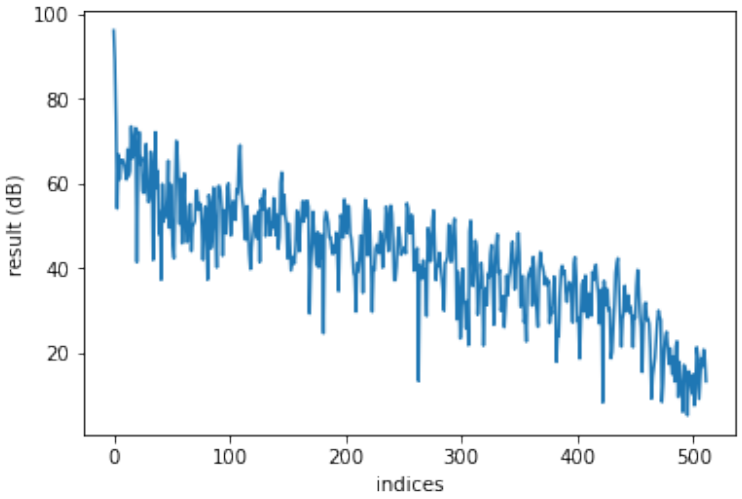


Figure 4.3: Example of result using an FFT

This result is similar to what one could get with specialized software (Audacity[4] for instance).

4.3.5 SPL determination for subbands

This part is described in the part 3.2.2 of the pdf available at:
http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf
It was only partially implemented because It requires data from external parts of the project not yet implemented.

[3] It's shiny...
[4] https://manual.audacityteam.org/man/plot_spectrum.html

4.3.6 Tonal and non-tonal components

To determine the masker, we need to find the tonal and non-tonal components. We determine them using the following method:

1. Determining local maxima, they satisfy this relation $SPL(k) > SPL(k+1)$ and $SPL(k) > SPL(k-1)$ with $k \in [1; N/2]$
2. Using the maxima, we check that they verify the following condition: $SPL(k) - SPL(k+j) \geq 7$ dB for all j in range:
   (a) if $2 < k < 63$ then for each value of j in $\{-2, 2\}$, the condition must be satisfied.
   (b) if $63 \leq k < 127$ then each of the previous j and the following one in $\{-3, 3\}$, the condition must be satisfied.
   (c) if $127 \leq k < 255$, then j must satisfy the previous conditions and for these values: $\{-6, 6\}$
   (d) if $255 \leq k \leq 500$, finally, we add: $\{-12, 12\}$
3. If the index k satisfies all these conditions, we add it to the list of tonal components.

For this part (and not only this one), we need an implementation of lists. We decided to go for a static list implementation, which follows the following declaration:

```
typedef struct static_list
{
    size_t *data; // stores indexes
    size_t size; // real size in memory
    size_t nb_el; // number of elements
} static_list;
```

Figure 4.4: Declaration of the *static_list* type

We implemented the following operations:

1. Append including extensions if the array is full.
2. Pop at index i.
3. Contains to search an item in the list.

To find noise components in the samples, we use another method. This method scans through critical bands. These follow an ISO norm and are provided in this pdf (table 3.8):

http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf

Each critical band in the table represents a range of indexes in the table of sound pressures. We sum the pressures of the frequencies not yet treated by the previous function (tonal process).
As a friendly reminder, please remember that the decibel is not a linear scale. To sum it, we must convert the values back to powers, sum them, and compute the resulting sound pressure level[5].
The next step for this algorithm is to compute the center of frequency matched with the index of the noises. It is the result of a sum weighted by the critical bands:

$$center = \frac{\sum_{n=cbi}^{cbi+1} 10^{SPL(n)/10}(z(cb(j)) - i)}{10^{power/10}}$$

Where power is the addition of each sound pressure, cbi and $cbi + 1$ are the ranges of the critical bands in the array of volumes. We round them to get the associated index. z is a function that provides the critical bandwidth associated with frequency f.

Finally, we get two lists with the tonal and non-tonal sounds. Those lists contain indices of the SPL array.

4.3.7 Decimation: remove useless sounds

As previously, our ears are not perfect. We cannot hear frequencies under a threshold in decibel, which varies for each frequency. Also, if two tonal sounds are too close, we only hear one.
We can compute the threshold of hearing thanks to this formula for each frequency f:

$$T(f) = 3.64\ (\frac{f}{1000})^{-0.8} - 6.5\ \exp{(-0.6(\frac{f}{1000} - 3.3)^2)} + 10^{-3}(\frac{f}{1000})^4\ (dB)$$

So for each tonal and non-tonal component, we check if the volume is above that threshold. If it is not the case, we discard it.

For tonal components, we need to compute their relative distance in barks. If $z(t[i+1]) - z(t[i]) < 0.5\ (bark)$ then we remove the one with the lowest volume according to the array of sound pressures. For the sake of simplicity in the formula, $t[index]$ represents the frequency associated with the index requested.

[5] See: https://www.wikiwand.com/en/Decibel

4.3.8 Masking thresholds for non-tonal and tonal components

We have now down-sampled the sound provided to a subset of tonal and non-tonal components. However, in reality, we can afford to have more data than this small subset. For a sampling rate at 44.1 kHz, we can have 130 ranges of frequencies with their masking threshold[6].
Rather than going through the numerous formulas, we will be giving a snippet of code to illustrate the process:

```
1     // for the 130 critical bands do...
2     for (size_t i = 0; i < crit->size; i++)
3     {
4         // initializes list of volumes
5         masks[i] = initStaticListF();
6         // critical band associated to frequency at index i
7         long double zi = crit->barks[i];
8         for (size_t j = 0; j < t->tonals->nb_el; j++)
9         {
10            // index of tonal component in the list
11            size_t k = t->tonals->data[j];
12            // map gives the nearest critical band from the
      frequency given by k
13            long double zj = crit->barks[map[k]];
14            long double dz = zi-zj;
15            if (dz >= -3 && dz <= 8)
16            {
17                long double avtm = -1.525 - 0.275 * zj - 4.5;
18                long double vf = 0;
19                if (dz >= -3 && dz < -1)
20                    vf = 17 * (dz+1) - (0.4 * SPL[k] + 6);
21                else if (dz >= -1 && dz < 0)
22                    vf = dz * (0.4 * SPL[k] + 6);
23                else if (dz >= 0 && dz < 1)
24                    vf = -17*dz;
25                else if (dz >= 1 && dz <= 8)
26                    vf = -(dz - 1) * (17 - 0.15 * SPL[k]) - 17;
27                // append the sum to the list
28                appendStaticListF(masks[i], SPL[k]+vf+avtm);
29            }
30        }
31    }
```

Figure 4.5: Masking threshold computation for tonal components

The resulting array of lists contains the different masking thresholds for each component.

[6]We cannot say it enough, please read http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf at 3.2.6

4.3.9 Global masking threshold

The global masking threshold is the sum of all thresholds (from tonal and non-tonal components) for each critical band. Again, the sum of decibels is not a regular sum. We must merge the values and sum them all together using a function. Our implementation is as follows:

```
1     // create a list of long floats
2     static_list_f *g_masks = initStaticListF();
3     // size = 130, the number of critical bands
4     for (size_t i = 0; i < size; i++)
5     {
6         // merge all values into one array
7         // its size is 1 (thr) + nb el in noise and tonal
8         long double *values = calloc(1+tonal[i]->nb_el+noises[i]->
      nb_el, sizeof(long double));
9         values[0] = table->thresholds[i];
10        // copy at offset 1
11        memcpy(values+1, tonal[i]->data, tonal[i]->nb_el*sizeof(
      long double));
12        // copy at offset 1+nb values in tonal
13        memcpy(values+1+tonal[i]->nb_el, noises[i]->data, noises[i
      ]->nb_el*sizeof(long double));
14
15        // computes the dB sum
16        long double final = add_db(values, 1+tonal[i]->nb_el+noises
      [i]->nb_el);
17        // appends it to the list of global masks
18        appendStaticListF(g_masks, final);
19        free(values);
20    }
21    return g_masks;
```

Figure 4.6: Global masking threshold computation

4.3.10 Minimum masksing threshold

Now, we must go back to our subbands. They are only 32 for this encoding process. To reduce the previous 130 samples, we get the minimum between each subband. In the C language, we get:

```
long double *getMinimumMaskThr(static_list_f *global, size_t *map)
{
    long double *mask = calloc(NB_SUBBANDS, sizeof(long double));
    for (size_t i = 0; i < NB_SUBBANDS; i++)
    {
        // gets the minimum between index first and last of global
    masks thr
        size_t first = map[i*SUB_SIZE];
        size_t last = map[(i+1)*SUB_SIZE -1];
        long double min = INFINITY;
        for (size_t i = first; i <= last && i < global->nb_el; i++)
        {
            if (global->data[i] < min)
                min = global->data[i];
        }
        // default value is 0
        mask[i] = min == INFINITY ? 0 : min;
    }
    return mask;
}
```

Figure 4.7: Minimum mask value for each subband

4.4 Conclusion

The psycho-acoustic model for the encoding is already quite vast. It is an old model but still illustrates the complexity of the science behind it.

According to Github, this part of the project represents around 1200 lines of code written by at most two person. It does not include the multiple tools and hours of research needed to understand the concepts behind the code. For instance, we coded the equivalent code in python to test our code and create a test suite.

Moreover, the MP3 standard was a proprietary codec[7] for a long time. It means that one cannot find easily on the Internet resources and standards. That led to lengthy research in the depth of Google to find decent documentation to do our implementation.

These difficulties, added to the level of complexity of this codec led to the abondonment of this part. We lacked of experience and started the work too late. This part of our project could be an entire project on a semester due to its complexity.

[7] https://www.wikiwand.com/fr/MP3 ISO standard: 11172–3, 13818–3

Chapter 5

Creating a new file Format

This part was done by Yabsira Alemayehu MULAT & Jean Bou Raad.

5.1 Huffman Coding

5.1.1 Introduction

Huffman coding is an efficient method of compressing data without losing information (data). It is an algorithm developed by David A. Huffman [1]while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

It provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as much will be encoded as longer strings. Therefore, this algorithm generates a code (Huffman code) to the message that will be encoded based on the frequencies of each symbols or characters in the message.

5.1.2 Huffman Compression Algorithm

The Huffman Compression algorithm has four main processes. These processes are:

1. Building a frequency list for each symbol or characters in the file.
2. Building the Huffman Tree which contains the codes for every symbol..
3. Encoding the data from the Huffman tree.
4. Compressing the file.

[1] https://www.wikiwand.com/en/Huffman_coding#overview

Building a frequency list

The frequency list is a list that contains the frequencies of each and every character in the file. The algorithm is straight forward[2]. First we go through the data which is bunch of strings. Next, we count the occurrence of each character and save it as their frequency values. Finally, we build a list of every symbols or characters with their respective frequencies and return the list.

The implementation is as follows:

```
1  struct glist *build_frequency_list(char *str, size_t len)
2  {
3      glist *result = init_glist();
4      // Initialize a list of zeros
5      size_t dataList[HUFF_MAX] = {0};
6      for (size_t i = 0; i < len; i++)
7      {
8          // Increment the frequencies of each chars
9          dataList[(unsigned char)str[i]] += 1;
10     }
11     for (size_t j = 0; j < HUFF_MAX; j++)
12     {
13         if (dataList[j] != 0)
14         {
15             // Tuple structure
16             freq_info f = {
17                 .item = j,
18                 .frequency = dataList[j],
19             };
20             //append all the tuples of values and frequencies of
       each character
21             append_glist(result, f);
22         }
23     }
24     return result;
25 }
```

Figure 5.1: Building frequency list

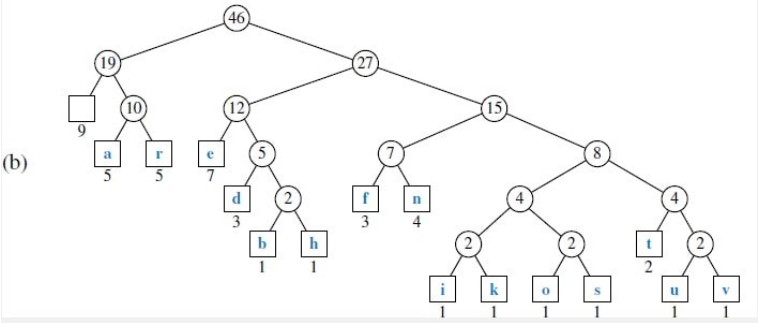
To give more clarifications on the code above, the **HUFF_MAX** is a Macro for the range of ascii values we expect from the data of the file. The **struct glist** is a structure that we built for lists which contain attributes like data, number of elements and size of the list.

[2] https://www.wikiwand.com/en/Huffman_coding#Terminology

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

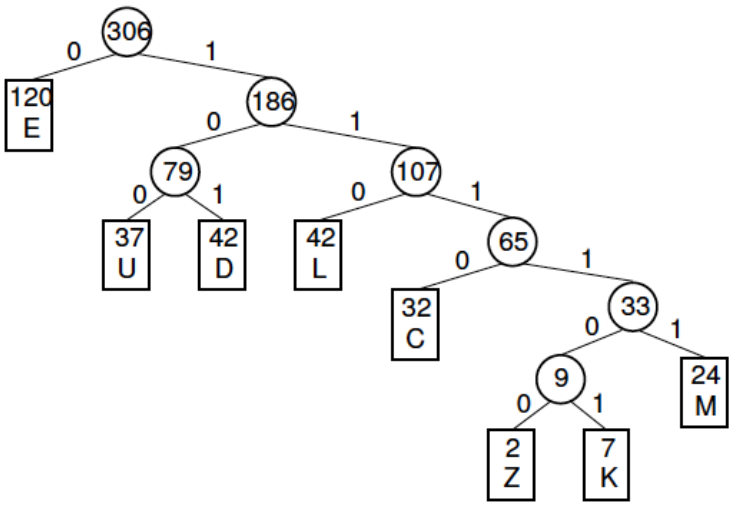
(b)



Frequency list

Building the Huffman Tree

The second step in the Huffman coding algorithm is building the Huffman tree. The Huffman tree is a binary tree that is build from the frequency list. Basically, by taking the frequency list as an input, we build a tree by giving priorities to the characters having a less frequency. We create nodes for the symbols or characters with less frequency and build the tree until all the characters are part of the tree. The function takes the frequency list as an input and returns binary tree.



A Sample of the Huffman tree

The implementation is as follows:

```
1  binTree *buildHuffmanTree(glist *freq_list)
2  {
3      tlist *t = init_tlist();
4      for (size_t i = 0; i < freq_list->nb_el; i++)
5      {
6          binTree *tree = createTree(freq_list->data[i], NULL, NULL);
7          append_tlist(t, tree);
8      }
9      decreasing_sort(t);
10
11     while (t->nb_el > 1)
12     {
13         binTree *freq1 = tree_pop(t);
14         binTree *freq2 = tree_pop(t);
15         freq_info f = {
16                 .item = 0,
17                 .frequency = freq1->key.frequency + freq2->key.
       frequency,
18         };
19         binTree *new = createTree(f, freq2, freq1);
20         // just in case to check for real root cause we have null
       character issue
21         new->flag = 1; // 1 indicates a non terminal node
22         append_tlist(t, new);
23         decreasing_sort(t);
24     }
25     binTree *result = t->data[0];
26     if (!result->left && !result->right)
27         result->flag = 0;
28     free(t->data);
29     free(t);
30     return result;
31 }
```

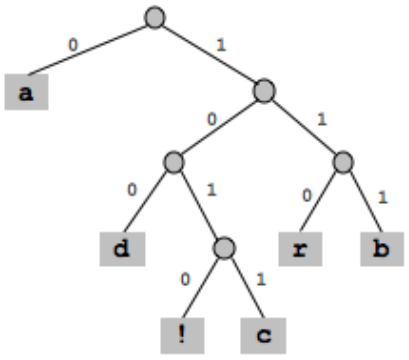
Figure 5.2: Building Huffman Tree

In the code above, there are different tree operations like **createTree**, **tree_pop** and also the **binTree** structure. The **createTree** is a function that creates a binary tree using the root, left, and right nodes. The **tree_pop** is a function that allows to pop an element from a binary tree with an index. And finally the **binTree** is a structure that contains the essential elements of a binary tree such as the key, left, and right values. There is also a sorting function which is the **decreasing_sort()** that sorts the keys of a tree in descending order[3].

[3] https://www.wikiwand.com/en/Huffman_coding#Problem_definition

Encoding the tree

The third step for compressing the file is to encode the data of the tree. Before compressing a file, the data of a file is composed of characters or symbols that consists of 8 bits of memory according to their specific ASCII values represented in binary numbers. Therefore, the reason we built the Huffman tree is to generate a Huffman code from the tree. These codes have less number of bits compared to their original size in the original file.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Encoding Tree

In the picture above you can see that there are Os and 1s in every edge between all the nodes. All the edges in the right part of the sub-trees have 0s and all the left sub-trees have 1s. Therefore, to generate a code for a single character, we traverse through the tree and concatenate the 0s and 1s until we reach the character.

The implementation is as follows:

```
1  char *encodeDataHuff(binTree *tree, char *str, size_t len)
2  {
3      // array of strings, one cell per ascii code
4      char *charcode[256] = {0};
5      GString *result = g_string_new("");
6      char *res;
7      for (size_t i = 0; i < len; i++)
8      {
9          // if we haven't found yet the associated code
10         if (!charcode[(unsigned char )str[i]])
11         {
12             //size_t len = 0;
13             string *s = malloc(sizeof(string));
14             s->data = malloc(129*sizeof(char));
15             s->allocated = 129;
16             s->len = 0;
17             __occurence(tree, str[i], s, 0);
18             // keeps it in the array now
19             charcode[(unsigned char )str[i]] = s->data;
20             //free(s->data);
21             // we free the string structure but not the data
       because we need it
22             free(s);
23         }
24         g_string_append(result, charcode[(unsigned char) str[i]]);
25     }
26     for (size_t i = 0; i < 256; i++)
27     {
28         if (charcode[i])
29         {
30             free(charcode[i]);
31         }
32     }
33     res = result->str;
34     g_string_free(result, FALSE);
35     return res;
36 }
```

Figure 5.3: Encode data

```
1  void __rec(binTree *tree, GString *result)
2  {
3      if (!tree)
4          return;
5      if (!tree->flag)
6      {
7          g_string_append(result, "1");
8          char *s = ascii_to_binary(tree->key.item);
9          g_string_append(result, s);
10         free(s);
11     }
12     else
13     {
14         g_string_append(result, "0");
15         __rec(tree->left,result);
16         __rec(tree->right,result);
17     }
18 }
19
20 char *encodeTree(binTree *tree)
21 {
22     GString *result = g_string_new("");
23     __rec(tree, result);
24     char *res = result->str;
25     g_string_free(result, FALSE);
26     return res;
27 }
```

Figure 5.4: Encode tree

Compressing the file

Before going directly to the algorithm of compression, there will be an explanation about the data type of the file which will be compressed in our project. In wav format audio files, the data is stored in the wav file is uncompressed raw file composed of the header and the data. In order to compress this file first we have to convert it to binary numbers. The result that we get from the encode data is a binary number. Therefore, finally we convert this binary numbers into **bitstreams** to get 8 bit binary values. We created different functions to perform different operations on this bitstreams[4].

The structure of the bitstream contains the following:

```
typedef struct bitStream
{
    char *data; // stream of data
    size_t offset_w; // offset for writing purposes per bit
    size_t cell_w; // offset in the cell % 8
    size_t len_r; // maximum reading index
    size_t offset_r; // maximum offset in cell % 8 for len_r
    size_t allocated_len;
} bitStream;
```

Figure 5.5: Structure of the bitStream

Some of these operations are:

[4] https://www.wikiwand.com/en/Huffman_coding#Compression

```
// appends a bit to a bitstream with realloc if needed
void appendBitStream(bitStream *s, char el)
{
    // reallocation if needed
    if (s->allocated_len == s->offset_w)
    {
        s->data = realloc(s->data, sizeof(char)*s->allocated_len*2)
    ;
        s->allocated_len *= 2;
    }
    // if it is 0 we shift one time to the left (LMSB)
    if (el == '0')
    {
        s->data[s->offset_w] = s->data[s->offset_w] << 1;
        // cell offset update modulo 8 bits
        s->cell_w = (s->cell_w+1) % 8;
        if (!s->cell_w)
            s->offset_w++;
    }
    // if it is 1 we shift one time to the left (LMSB) & add one
    else if (el == '1')
    {
        s->data[s->offset_w] = s->data[s->offset_w] << 1;
        s->data[s->offset_w] |= 1;
        s->cell_w = (s->cell_w+1) % 8;
        if (!s->cell_w)
            s->offset_w++;
    }
    else
    {
        errx(EXIT_FAILURE, "unknown char");
    }
}
```

Figure 5.6: Initialization and Append operation on bitstreams

```
// encodes a full string of data, all chars are '0' or '1'
// except for null termination
bitStream *encodeData(char *s)
{
    bitStream *b = initializeBitStream();
    for (size_t i = 0; s[i]; i++)
    {
        // appends a 0 or a 1 to a bitStream
        appendBitStream(b, s[i]);
    }
    // updates padding according to the result
    b->offset_r = !b->cell_w ? 0 : 8 - b->cell_w;
    return b;
}
```

Figure 5.7: Encode data for bitstream

```
1  // slow decoding process o(n)
2  char *decodeBitStream(bitStream *stream)
3  {
4      GString *s = g_string_new("");
5      for (size_t i = 0; i < stream->offset_w; i++)
6      {
7          // convert char to bit representation
8          // can reduce allocation by using an unique buffer
9          char *base2 = toBitRepresentation(stream->data[i]);
10         g_string_append(s, base2);
11         free(base2);
12     }
13     char *last = toBitRepresentation(stream->data[stream->offset_w
       ]);
14     g_string_append(s, last+stream->offset_r);
15     free(last);
16     char *result = s->str;
17     g_string_free(s, FALSE);
18     return result;
19 }
```

Figure 5.8: Decode data and bit representation on bitstream

Now, having the data converted to bitstream, we can compress the file. We categorized the data into three parts. The categories are :

1. TREE
2. HDAT
3. DUPL

These categories are IDs for specific types of data. The **TREE** contains the Huffman tree, the **HDAT** contains the Huffman data and finally the **DUPL** contains duplicated characters used in the data. We can use those chunks for other purposes as well if needed.

Using these IDs we compressed the files by encoding the data in less number of bits from the result we got from encoding the data in the bitstream. Basically, we encoded all the data with respect to their IDs and created an output data with less size.

The method looks as follows:

```
void compressToFile(char *data, size_t len, char *output)
{
    // classic Huffman compression process
    glist *freq = build_frequency_list(data, len);
    binTree *t = buildHuffmanTree(freq);
    free_glist(freq);
    // data under the form of '0' and '1' (bytes)
    char *encoded_tree = encodeTree(t);
    char *encoded_data = encodeDataHuff(t, data, len);
    // To bitstream: from 8 bytes to 1 byte
    bitStream *b = encodeData(encoded_tree);
    // structure for the output
    output_h *tree = createOutput(b, "TREE");

    int fd = open(output, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    // writes the structure and the data to the file
    writeStructure(fd, tree);
    freeBitStream(b);
    free(tree);
    free(encoded_tree);

    bitStream *d = encodeData(encoded_data);
    tree = createOutput(d, "HDAT");

    writeStructure(fd, tree);
    close(fd);
    // Freeing zone
    freeBitStream(d);
    free(tree);
    freeBinTree(t);
    free(encoded_data);
}
```

Figure 5.9: Compression

5.1.3 Huffman Decompression Algorithm

The decompression process of a file is basically the translation of the encoded data from the Huffman tree to single characters and put them back to their previous ASCII values with 8 bits structure. To perform this algorithm, we created functions to **decode the Huffman data** and **decode the Huffman Tree**. By using this functions we were able to decompress the file to its original size.

The functions are implemented as follows:

```
// Slow integral decompression process o(n)
char *decodeDataHuff(binTree *tree, char *str, size_t len, size_t *
    total)
{
    string s =
    {
        .allocated = 512,
        .data = calloc(512, sizeof(char)),
        .len = 0,
    };
    binTree *t = tree;
    for (size_t i = 0; i < len; i++)
    {
        // if it is a leaf...
        if (!t->flag)
        {
            //  check if the string is full
            if (s.len == s.allocated)
            {
                // realloc if needed
                s.data = realloc(s.data, s.len * 2 * sizeof(char));
                s.allocated *= 2;
            }
            // append the leaf's item
            s.data[s.len] = t->key.item;
            s.len++;
            // reset the traversal
            t = tree;
        }
        if (str[i] == '0')
            t = t->left;
        if (str[i] == '1')
            t = t->right;
    }
    // last element
    s.data[s.len] = t->key.item;
    s.len++;
    *total = s.len;
    return s.data;
}
```

Figure 5.10: Decode Huffman data

```
struct tuple_bi __decodeTree(char *data, size_t i)
{
    // maybe not safe, check for strlen as param mb if errors
    if (!data[i])
    {
        struct tuple_bi res =
        {
            .t = NULL,
            .i = i,
        };
        return res;
    }
    if (data[i] == '1')
    {
        char key = fromBinToByte(data+i+1);
        freq_info f =
        {
            .frequency = 0,
            .item = key,
        };
        binTree *t = createTree(f, NULL, NULL);
        struct tuple_bi res = {
            .t = t,
            .i = i+8,
        };
        return res;
    }
    freq_info f = {
        .frequency = 0,
        .item = 0,
    };
    binTree *t = createTree(f, NULL, NULL);
    // check that it is correct
    t->flag = 1;
    struct tuple_bi res_l = __decodeTree(data, i+1);
    struct tuple_bi res_r = __decodeTree(data, res_l.i+1);
    i = res_r.i;
    t->left = res_l.t;
    t->right = res_r.t;
    struct tuple_bi final =
    {
        .t = t,
        .i = i,
    };
    return final;
}
```

Figure 5.11: Decode Huffman tree

[5]

[5] https://www.wikiwand.com/en/Huffman_coding#Decompression

5.2 Adaptation of WAVE file format

To compensate the removal of the MP3 encoding process in our book of specifications, we decided to create a new kind of compression.
This process is by itself a challenge because of the limited time we had.
We implemented compression & decompression algorithms for a new compressed format which we will explain here.

5.2.1 General encoding process

This part showcases the steps to encode PCM wave files into Huffman Compressed wave files.

Changes from classic WAVE file format

The WAVE file format is a modular file format. It offers many functionalities & can be adapted for new purposes. We can add new chunks and add new norms to existing ones. However, these add-on are supported only by our project because they do not fit into the norms.

The changes are:

1. Change to compression codes: we encode using a special encoding code: 42 which indicates H3rtz compression file. It differentiates the new files from the classic PCM ones.

2. Modified data chunk: the data chunk is cut into smaller parts because the data is Huffman compressed. Hence, we need the compression tree and the actual compressed data. **The compression is lossless.**

Remember: this specific compression can only be read by our project

Implementation

We can compress a PCM signal using the Huffman compression functions. The process can look trivial, however, it is not because we need to handle IO correctly. We must:

1. Write the header completely in a new file using the tools created by Kevin-Brian.

2. Compress the data & write it in the new file.

However, the first steps depends on the second one. But, the order we provided is the writing order we must follow.
To get around that inter-dependency, we output the compressed data to another file, which acts as a buffer, and then update the header. Finally, we can write correctly the new file.

Efficiency

Nowadays, with Free Lossless Audio Codec being a standard in the industry, lossless compression got much more efficient than what it was before. It can reach up to 50% file size reduction.
This codec uses multiple compression algorithms to reach that level of efficiency. Hence, using only one leaves only little optimization.

With Huffman compression, we can only reach 1% to 5% size reduction. Indeed, raw signals are not redundant most of the time. It decreases the efficiency of Huffman's compression algorithm. It is based on the fact that some values will appear more than others which is not usually the case sadly.

However, the algorithm fulfills its main purpose: showcase our understanding of the WAVE file format & general encoding conventions.

Finally, the compression is fast. For 80 millions bytes, the compression is done in 6 seconds on a single thread. Of course, performance varies from one setup to another so please take those numbers with precautions.

5.2.2 Decoding

The decoding process required no adaptation of our header parser. Indeed, the only conventions we changed is inside the DATA chunk. It does not affect the parser.

Algorithms: slow like a turtle

Decompression at least a few seconds to complete using the algorithms implemented. This time is acceptable when decompressing the entire file to go back to original PCM. However, it is not when using it for playback: the loading takes too much time and freezes the player completely. We needed to make it faster. We tried first to decode the data by chunks of 512 or more but it ended up being too complicated for implementation.

The solution adopted bypasses many steps of the original algorithm to reduce its complexity.

What we did for decompression is transforming the stream of bits into a string of '0' and '1'. Doing this step before running the algorithm could be an option. However, its complexity is $O(n)$[6] and takes 8 times more memory than the bitstream representation. To bypass this conversion, we implemented bit by bit reading directly on the bitStream. Hence, when a bit needs to be accessed and

[6] n is the string's length

converted into a '0' or '1', it can be done in $O(1)$ without any new memory usage.

From this step, we added character by character decoding which avoids the usual $O(n)$ complexity to retrieve the whole data. Retrieving a single character is done using a traversal guided by the data which can take at most $O(H)$ (H is the Huffman's tree height) in worst case scenario. The decoding for a single character looks like this:

```
1  // @param len: length of the encoded data
2  // @param total: size_t pointer to the total size of the data
3  char decodeDataHuffSingle(binTree *tree, bitStream *b, size_t *r,
4                            size_t *o, uint8_t *possible)
5  {
6      binTree *t = tree;
7      for (size_t i = 0; *r < b->offset_w; i++)
8      {
9          // if it is a leaf...
10         if (!t->flag)
11         {
12             *possible = 1;
13             break;
14         }
15         // retrieves a single bit from the BitStream
16         char d = decodeIndex(b, r, o);
17         if (d == '0')
18             t = t->left;
19         if (d == '1')
20             t = t->right;
21         // updates for reading indexes, increments by one
22         *o = (*o + 1) % 8;
23         !*o ? (*r)++ : *r;
24     }
25     if (*r == b->offset_w && t->flag)
26         *possible = 0;
27     return t->key.item;
28 }
```

Figure 5.12: Decoding function

5.2.3 Linking to PulseAudio - Back-end

Adapting PulseAudio to instantaneous decoding is not an easy task. It requires heavy modifications to the backend to ensure good compatibility with the rest of the code.
To make synchronous decompression work, we must:

1. Prepare the data: first decode the entire Huffman tree and put into the BitStream structures the compressed data

2. Decompress the data when needed: this task requires the tracking of indices to read the right data at the right moment.

The first step can be done using very simple functions. Indeed, they are completely independant from the rest of the backend.

```
// loads a tuple with the Huffman tree + the data under the
    structure BitStream
tuple decodeHuffmanPart(wav *w)
{
    // Tuple for the result
    tuple t = {0};
    if (!w || w->fmt->AudioFormat != h3rtz)
        return t;
    size_t i = 0;
    // restores the original size in the header
    size_t original_length = loadOnlyDUPL((char *)w->data->chunk, &
    i);
    // Decode the Huffman tree
    binTree *tree = decompressTree((char *)w->data->chunk, &i);
    // Puts the compressed data in a bitStream structure
    bitStream *b = loadOnlyData((char *)w->data->chunk, &i);
    w->data->data_bytes = original_length;
    t.a = tree;
    t.b = b;
    return t;
}
```

Figure 5.13: General process to prepare decompression

The data given by this function is then stored in a structure:

```
typedef struct huff_decode
{
    binTree *tree;
    bitStream *b;
    // indexes for reading purposes
    size_t r;
    size_t o;
} huff_decode;
```

Figure 5.14: Huffman decoding structure

The second steps uses the structure to read and feed PulseAudio's buffer in a similar way to the code in part 2.4.5.
We use more functions to feed the buffer, the main part being as follows:

```
1         if (flag) // flags the compressed format
2         {
3             uint8_t possible = 0;
4             // loop which writes at most bytes_to_fill bytes
5             for (i = 0; i < bytes_to_fill && i + wav->timing->
      offset < filelength; i += 1)
6             {
7                 possible = 0;
8                 // Decompression of data using r,o as index
9                 buffer[i] = (uint8_t)decodeDataHuffSingle(dec->tree
      , dec->b,&(dec->r), &(dec->o), &possible);
10                // possible indicates end of data
11                if (!possible)
12                    break;
13            }
14        }
```

Figure 5.15: Snippet of code from the writing callback function

This code has been optimized to avoid stuttering during playback. Previous versions of the decompression were decoding the needed data and then copying it to a buffer.
The following shows the worst case complexity of each implementation in different context. n is the data length, t is the tree's code length, H is the Huffman's tree height, S is the number of samples requested, it is generally a number in the ranges of 10000 to 50000. A quick reminder that $t << n$ so $O(t) << O(n)$ which means that decoding a tree is quicker than decoding the whole data.

Context	Original	Optimization 1	Optimization 2
Loading	$O(n+t)$ (Freezes GUI)	$O(t)$ (Fast)	$O(t)$ (Fast)
Playback	$O(S)$ (Fast)	$O(S(H+1))$ (Stutters)	$O(SH)$ (Fast)

The integration of the new file format was for most of the work about optimization. It allowed us to see the difference in real life that can make bad code with bad complexity and code with little complexity. The playback is currently smooth and uses little memory in excess compared to the classic PCM signal and uses around 1% of CPU during decompression (which is also playback). It is within industry's standard and proves that our algorithms are very efficient.

5.2.4 Linking with GUI

We created wrapping functions to connect both parts of the project. They take-in paths (input & output). The GUI, using file choosers, is able to retrieve both paths and give them to those functions. Using the WAVE headers, we can determine if the program must decompress or compress the file (with compression codes).

Chapter 6

GUI

6.1 Introduction

In the previous defenses, some basic features of the user interface such as the play-pause buttons, and also some advanced features such as the playlist had been implemented. The interface presented during the second defense looks similarly to the final product that H3rtz.stdio created this final defense, we were able to integrate important features such as audio converter and encoder and created more dialog to warn and guide the user of the interface if it is not used properly.

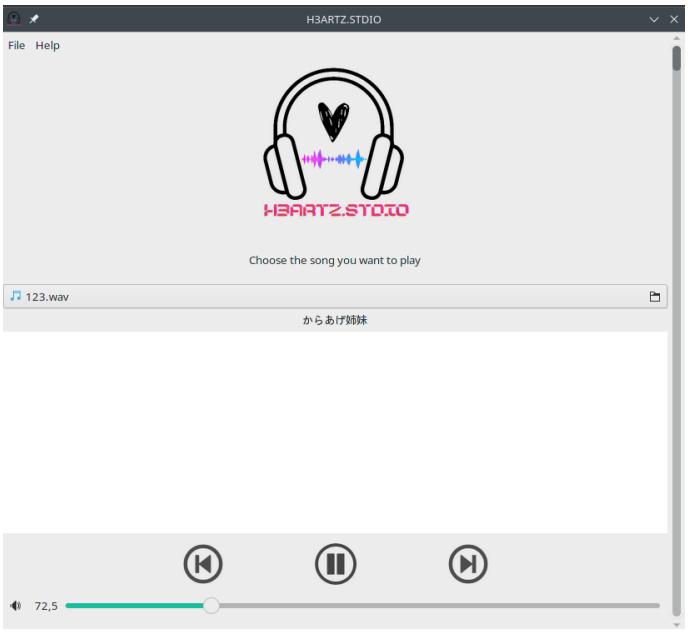
In the following part, all the important features are explained and showed in a chronological order, which means that the changes will represent the evolution of the interface during the project. It also means that it is completely normal to see different versions of the interface, and reading it in order will make more sense.

Finally, there will be a conclusion part to give a summary about what is done during this whole project.
[1]

[1] https://developer.gnome.org/gtk3/stable/

6.2 Glade

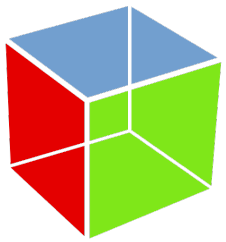
Glade is a RAD tool to enable quick & easy development of user interfaces for the GTK toolkit and the GNOME desktop environment.
The user interfaces designed in Glade are saved as XML, and by using the GtkBuilder GTK object these can be loaded by applications dynamically as needed.
For our project, we used this tool to design the following Interface. Glade has many features that helps designing different dynamic interfaces. Since our project is an audio player, we used most of the features like volume button (slider), play button, pause button, progress bar(slider), signals to connect the GTK to Glade object ...etc. The details of this interface will be given below.



For more information about Glade and it's documentations, click here:

https://glade.gnome.org/

6.3 GTK



GTK is a free and open-source cross-platform widget toolkit for creating graphical user interfaces. It will be used with Glade to create the GUI, specially to link the functions to the components of the interface.

6.4 The first defense

Features listed bellow in this section are the features implemented before the first defense.

6.4.1 File chooser

The file chooser component of the Interface design allows users to pick any audio file that they want to play. To make this work, GTK provides a function called **gtk_file_chooser_get_filename(GtkWidget *widget)** that allows to get files from a directory using the widget used in the interface design. To be more precise, the code is as follows:

```
1  void on_choose_wav(GtkWidget *widget, gpointer userdata)
2  {
3      static uint8_t init = 0;
4      gtk_player *player = userdata;
5      if (player->filename)
6          free(player->filename);
7      player->filename = gtk_file_chooser_get_filename(
       GTK_FILE_CHOOSER(widget));
8      if (!player->filename)
9          return;
10     int code = terminateStream(player->player);
11     if (code >= 0 && player->ui.ID)
12         g_source_remove(player->ui.ID);
13
14     parseFile(player->player, player->filename);
```

```
15      changeTitle(player);
16      if (prepareStream(player->player) == -1)
17          on_quit(NULL, player);
18      if (!init)
19      {
20          init = 1;
21          setDefaultVolume(player);
22      }
23      player->player->utility->current = 0;
24      on_play(player);
25  }
```

6.4.2 Play and Pause button

The pause and play button is a toggle button. Indeed, when the button is clicked, the icon changes. For that some icons have been designed, shaded and background removed to make the toggle as smooth as possible. And of course, this play and pause button is related to the playing music, the progress bar. For convenience and performance, our team used the built-in function **g_timeout_add()** instead of multi-threading.

```
1   void on_play(gtk_player *g)
2   {
3       if (g->player->pa_state != PAUSED && g->player->pa_state !=
        READY)
4           return;
5       if (!g->player->pulseAudio->stream)
6           return;
7       GtkWidget *image = gtk_image_new_from_file("./GUI/callbacks/
        icons/pause.png");
8       gtk_button_set_image (GTK_BUTTON(g->ui.play_pause), image);
9       play(g->player);
10      g->ui.ID = g_timeout_add(100, slider, g);
11      return;
12  }


1
2   void on_pause(gtk_player *g)
3   {
4       if (g->player->pa_state != PLAYING)
5           return;
6       GtkWidget *image = gtk_image_new_from_file("./GUI/callbacks/
        icons/play.png");
7       gtk_button_set_image(GTK_BUTTON(g->ui.play_pause), image);
8       Pause(g->player);
9       if (g->ui.ID)
10          g_source_remove(g->ui.ID);
11      g->ui.ID = 0;
12      return;
13  }
```

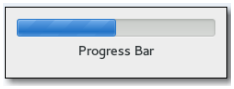
6.4.3 Progress Bar

As one guess from the name, the progress bar helps us to show the progress of a playing audio. It gives information about the time or duration of the audio. To implement this, there are two methods that we used. The methods are : -

- 1. GtkProgress Bar
- 2. GtkScale Button(slider)

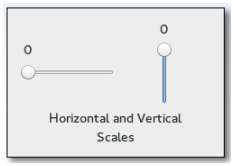
GtkProgress Bar

The GtkProgress Bar is a widget which indicates progress visually. It shows the progress of some process using percentages. For our project, we implemented the GtkProgress Bar but for the sake of dynamics the interface, we used the GtkScale Button(slider) which will be explained after to make it user friendly..



GtkScale Button(slider)

The GtkScale Button is a button which belongs to the GtkScale class that is a slider widget for selecting a value from a range. This allows users to click and go at any point of the scale. For our project, we implemented the GtkScale Button(slider) using another powerful tool from GTK which is the **gint g_timeout_add (guint32 interval, GtkFunction function, gpointer data);** function. This function is helpful to execute the function parameter within a time interval of the interval parameter in the function.

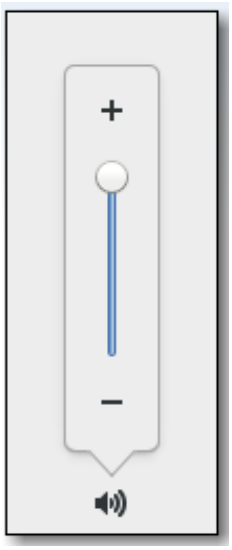


For more information about the documentations, click here:

- https://developer.gnome.org/gtk3/stable/GtkProgressBar.html
- https://developer.gnome.org/gtk3/stable/GtkScale.html
- https://developer.gnome.org/gtk3/stable/GtkAdjustment.html

6.4.4 Volume Button

In this section, we implemented a button that controls the volume of an audio with the help of GtkVolume Button and the information given from the pulseaudio which contains the volume. Since the GtkVolume button functions as a slider, we had to modify the information of the volume from the pulseaudio so as to get the value of the scale.



The implementation is as follows: -

```
1  void cvolume(GtkWidget *widget __attribute__((unused)), gpointer
       userdata)
2  {
3      gtk_player* player = userdata;
4      if (player->player->pa_state >= DRAINED)
5          return;
6      gdouble value = gtk_scale_button_get_value(GTK_SCALE_BUTTON(
       player->ui.volume));
7      player->player->info->volume = value;
8      setVolume(player->player);
9      return;
10 }
```

6.4.5 Name of artists

As the user choose the song to play in the designed interface, a component called GtkLabel will display the name of the artist if it exists, otherwise it will just display "Unknown".

The implementation is as follows:

```
1  void changeTitle(gpointer userdata)
```

```
{
    gtk_player *player = userdata;
    fileInfo *info;
    info = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->name, info->artists ? info->artists
    : "Unknown");
    free(info);
}
```

For more information about the documentations, click here:

https://developer.gnome.org/gtk3/stable/GtkLabel.html

6.5 The second defense

6.5.1 Information on songs

In this sub-part of the User Interface, one can find bellow the defined structures that will be used in later shown functions. They will give a better understanding.

```
typedef struct playlist_t
{
    pthread_t *threads;     // check if the task is finished
    wav **w;                // headers
    file **f;               // files IO
    size_t nb_el;           // nb of elements currently
    size_t size;            // total size of the list in memory
    size_t index;
} playlist_t;

typedef struct UserInterface //To avoid global variables (widgets)
{
    GtkWindow* window;
    GtkButton* play_pause;
    guint ID;
    GtkVolumeButton *volume;
    GtkScale *slider;
    GtkAdjustment *adjustment;
    char *name_chooser;
    GtkListStore *dialog_list_store;
    GtkTreeView *dialog_tree;
    GtkTreeSelection *select;
    GtkLabel *name;
    GtkLabel *genre;
    GtkLabel *album;
    GtkImage *song_image;
    GtkFileChooser *audio_chooser;
} UserInterface;

typedef struct gtk_player   //Main structure to access data
{
    char *filename;
    pa_player *player;
    file *data;
    UserInterface ui;
    playlist_t *playlist;
```

```
37 } gtk_player;
```

In this defense, H3rtz.stdio worked on making a better looking User Interface. For that, the team chose to display the most important information which are :

1. Album
2. Genre
3. Artist4
4. Name

The next sub-subsections will give more details about these features. But first, here is a picture that quickly shows the features of the interface that will be explained soon in the next sub-subsections.

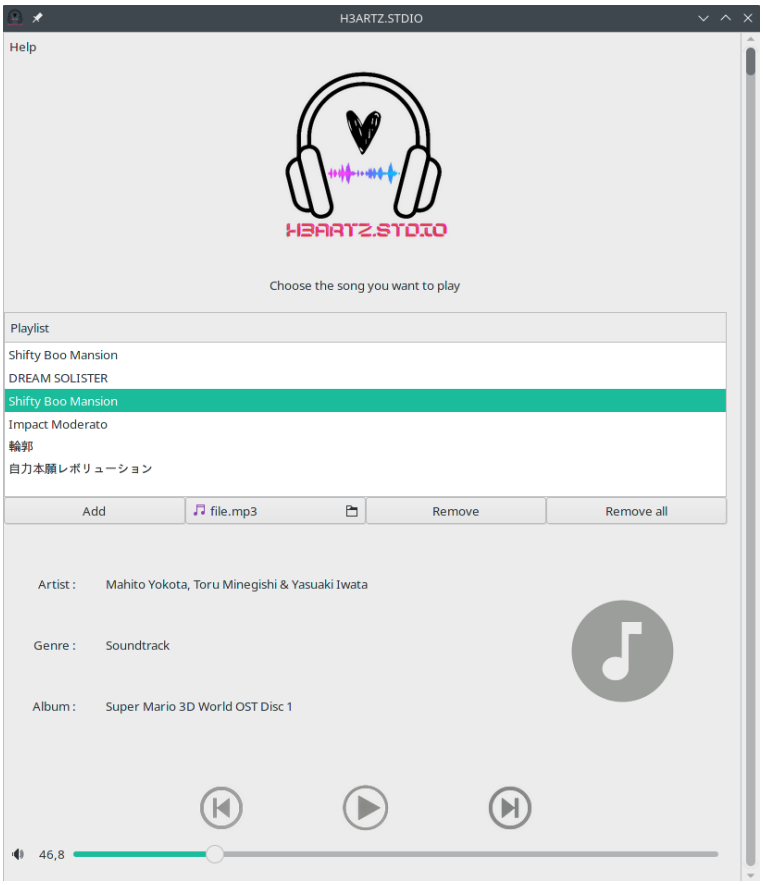


Figure 6.1: Main function of the album getter

6.5.2 Album of the song

The used data are obtained by parsing with a function that transforms the information of the header into an object of strings. When they are retrieved, to access the album, the code checks whether that specific info is given or not. If it is, the right album will be displayed in the label GtkLabel *album. Otherwise, it will display "Unknown".

```
void album(gpointer userdata)
{
    gtk_player *player = userdata;
    fileInfo *info;
    info = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->ui.album, info->album ? info->album
    : "Unknown");
    free(info);
}
```

Figure 7: Main function of the album getter

6.5.3 Genre of the song

This part works just like the previous part, but here the genre will be displayed in the GtkLabel *genre.

```
void genre(gpointer userdata)
{
    gtk_player *player = userdata;
    fileInfo *genre;
    genre = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->ui.genre, genre->genre ? genre->
    genre : "Unknown");
    free(genre);
}
```

Figure 8: Main function of the genre getter

6.5.4 Artist of the song

The data is accessed in the same way as the previous part. To access the artist of the playing song, the code checks whether that specific info is given or not. If it is, the artist will be displayed in the label GtkLabel *name. Otherwise, it will display "Unknown".

```
void changeTitle(gpointer userdata)
{
    gtk_player *player = userdata;
    fileInfo *info;
    info = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->ui.name, info->artists ? info->
    artists : "Unknown");
    free(info);
}
```

Figure 9: Main function of the artist getter

6.5.5 Name of the song

It is important to precise that the "name of the song" is not the display on the file chooser. It is actually given in the header, and retrieved by the function getFileInfo(), then accessed bellow with f→name. The displayed code bellow checks whether that specific info is given or not. If it is, the song's name will be displayed in the GtkTreeView (later explained in the section 4.3.1). Otherwise, it will display the path to the song.

```
1  void append(GtkWidget *widget __attribute__((unused)), gpointer
       userdata)
2  {
3      gtk_player *player = userdata;  // Initialization of player
       structure
4      GtkTreeIter iter;               // Value that hold the
       addresses of list items
5      gchar *str = player->ui.name_chooser; // Name of the file
       chosen
6      player->ui.dialog_list_store = GTK_LIST_STORE(
7          gtk_tree_view_get_model(player->ui.dialog_tree)); //
       Getting the model from glade
8      tuple data = ParseTrack(str);   // Gets the info about the
       track
9      if (!data.a || !data.b)
10         return;
11     player->playlist->f[player->playlist->nb_el] = data.a;
12     player->playlist->w[player->playlist->nb_el] = data.b;
13     player->playlist->nb_el++;
14     gtk_list_store_append(player->ui.dialog_list_store, &iter); //
       Appending elements to the list
15     wav *w = data.b;
16     fileInfo *f = getFileInfo(w->list);
17     char *entry = f->name ? f->name : str;  // Getting the filename
18     free(f);
19     gtk_list_store_set(player->ui.dialog_list_store, &iter,
       LIST_ITEM, entry, -1);
20     // Sets the value of one or more cells in the row referenced by
        iter
21 }
```

Figure 10: Main function of the song's name getter

6.5.6 Playlist

Before starting directly explaining about the implementation process we will introduce the tools that we used for this feature. Mainly, there are two tools that we used for this feature which are pulseaudio and GTK TreeView widget from GTK. In the previous report we gave brief introduction about pulseaudio and GTK. Hence, we will give the links to get more information about these two tools below[1] [2].

GTK TreeView

The GTK TreeView [3] is a widget used for displaying both trees and lists. It is part of the GTK Container tools hierarchically. To use this widget, we need to define a data structure. By data structure, it means either the lists or trees. For our project and specifically this defense, we used the list data structure. To define this structure in GTK, we used the GTK ListStore [4] which allows us to input a list inside a tree view build by the GTK TreeView. The ListStore structure contains different features like adding rows and columns, and with the integration of GTK TreeView, it makes items of a list clickable. It also helps the user to modify the items of a list.

We used two different structures of lists to take care of adding, removing elements and the stream of the loaded audio files. To be more precise, the GTK part is performing addition and removal of items from the list store while the other list structure concerned with PulseAudio is controlling the stream, moving to the next song or previous song and so on. These processes are running at the same time.

Basically, using the GTK TreeView widget we were able to add and remove different items in a list which is really important for the implementation of a playlist. The functions implemented are as follows: -

Adding item

```
1 void append(GtkWidget *widget __attribute__((unused)), gpointer
    userdata)
2 {
3     gtk_player *player = userdata;    // Initialization of player
    structure
4     GtkTreeIter iter;                 // Value that hold the
    addresses of list items
5     gchar *str = player->ui.name_chooser;    // Name of the file
    chosen
6     player->ui.dialog_list_store = GTK_LIST_STORE(
7         gtk_tree_view_get_model(player->ui.dialog_tree)); //
    Getting the model from glade
```

[1] https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/
[2] https://developer.gnome.org/gtk3/stable/
[3] https://developer.gnome.org/gtk3/stable/GtkTreeView.html
[4] https://developer.gnome.org/gtk3/stable/GtkListStore.html

```
8       tuple data = ParseTrack(str);  // Gets the info about the track
9       if (!data.a || !data.b)
10          return;
11      player->playlist->f[player->playlist->nb_el] = data.a;
12      player->playlist->w[player->playlist->nb_el] = data.b;
13      player->playlist->nb_el++;
14      gtk_list_store_append(player->ui.dialog_list_store, &iter);  //
         Appending elements to the list
15      wav *w = data.b;
16      fileInfo *f = getFileInfo(w->list);
17      char *entry = f->name ? f->name : str;   // Getting the
        filename
18      free(f);
19      gtk_list_store_set(player->ui.dialog_list_store, &iter,
        LIST_ITEM, entry, -1);
20      // Sets the value of one or more cells in the row referenced by
         iter
21  }
```

Figure 11: Function to add a song to the playlist

Removing item

```
1   void remove_item(GtkWidget *widget __attribute__((unused)),
        gpointer userdata)
2   {
3       gtk_player *player = userdata;   // Initialization of player
        structure
4       GtkTreeIter iter;               // Value that hold the
        addresses of list items
5       GtkTreeModel *model;
6
7       model = gtk_tree_view_get_model(player->ui.dialog_tree);  //
        Getting the model from glade
8       player->ui.select = gtk_tree_view_get_selection(player->ui.
        dialog_tree);   // Getting the GTK TreeSelection from glade
9
10      if (gtk_tree_model_get_iter_first(model, &iter) == FALSE) //
        Checking if the list is empty
11          return;
12
13      gboolean found = gtk_tree_selection_get_selected(player->ui.
        select,
14                                                       &model, &iter)
        ;
15      if (!found)
16          return;
17      GtkTreeIter iter_bis;
18      if (gtk_tree_model_get_iter_first(model, &iter_bis) == FALSE)
19          return;
20      gchar *key;
21      gtk_tree_model_get(model, &iter, LIST_ITEM, &key, -1);
22      ssize_t l = findIndex(player, key); // Getting the index of the
         songs in the list
23      if (l == -1)
24          return;
```

```
25      size_t i = (size_t)l;
26      if (i >= player->playlist->nb_el)
27          return;
28      if (i == player->playlist->index)
29      {
30          Pause(player->player); // Pause the audio
31          terminateStream(player->player);  // Terminate the stream
32      }
33      removeTrackAtIndex(player->playlist, i); // Remove from the
        PulseAudio list structure
34      gtk_list_store_remove(player->ui.dialog_list_store, &iter); //
        Remove from the Gtk ListStore
35  }
```

Figure 12: Function to remove a song from the playlist

Removing all items

```
1   void remove_all(GtkWidget *widget __attribute__((unused)), gpointer
        userdata)
2   {
3       gtk_player *player = userdata; // Initialization of player
        structure
4       GtkTreeModel *model;
5       GtkTreeIter iter; // Value that hold the addresses of list
        items
6
7       model = gtk_tree_view_get_model(player->ui.dialog_tree);  //
        Getting the model from glade
8
9       if (gtk_tree_model_get_iter_first(model, &iter) == FALSE)
10      {
11          return;
12      }
13
14      gtk_list_store_clear(player->ui.dialog_list_store); // Remove
        from GTK ListStore
15      if (player->player->pa_state == ACTIVE)
16      {
17          Pause(player->player);
18          terminateStream(player->player);
19      }
20      cleanPlaylist(player->playlist); // Remove from the PulseAudio
        list structure
21  }
```

Figure 13: Function remove all song from the playlist

Playlist Design samples

In this project we designed two layouts on how to show the playlist. The image samples are below :

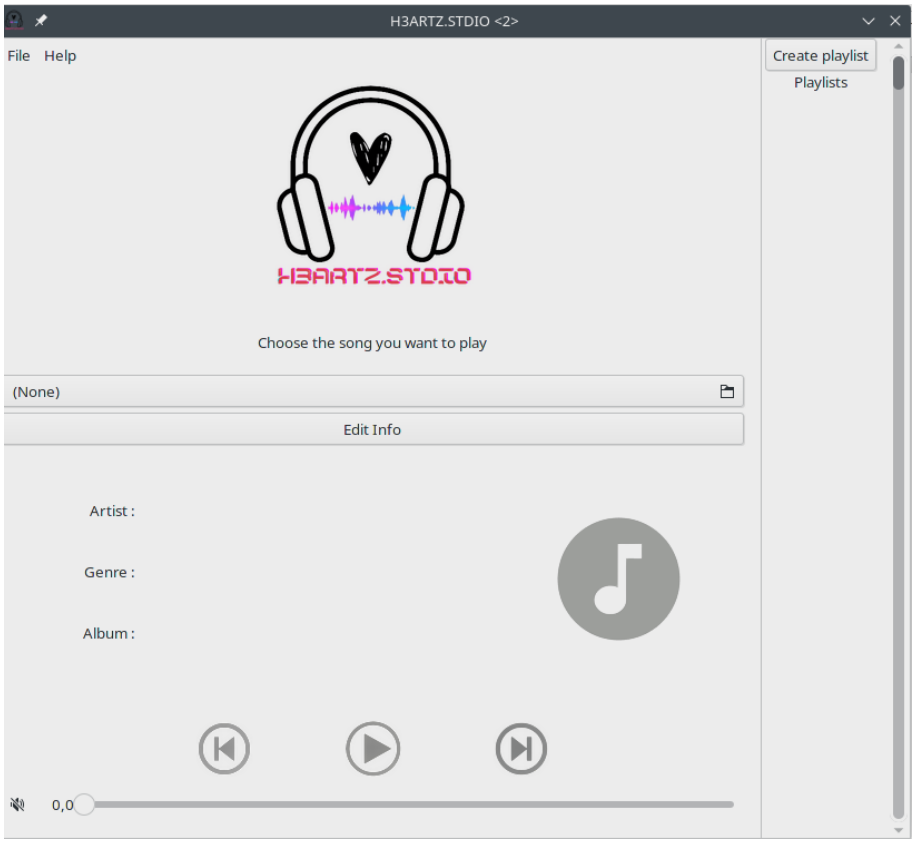


Figure 15: First version of the playlist's display

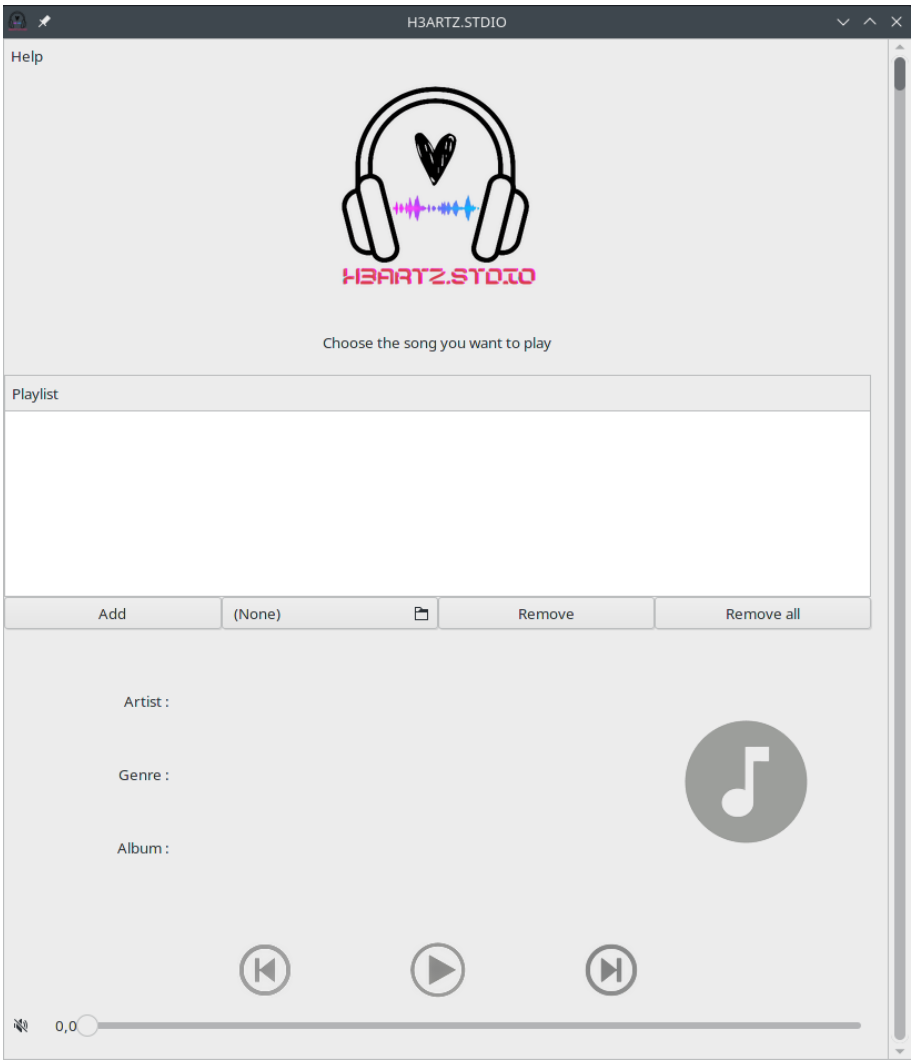


Figure 16: Second version of the playlist's display

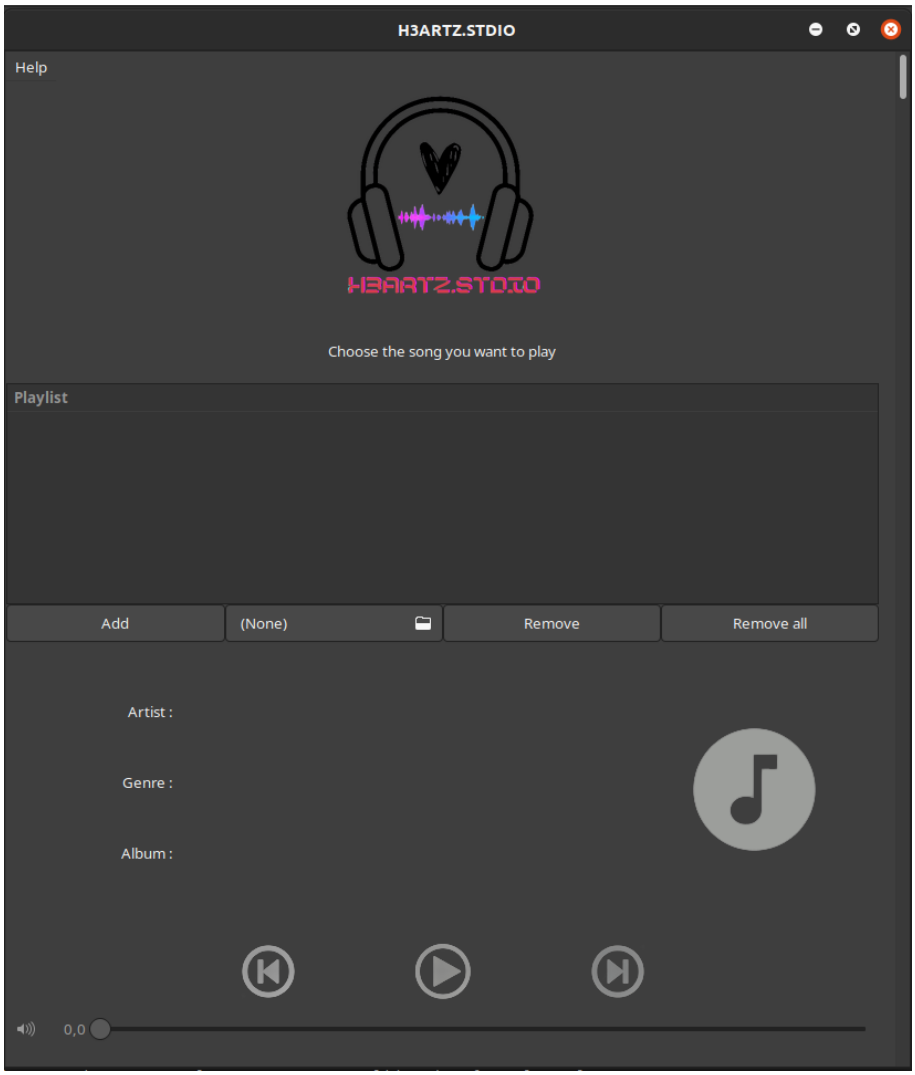


Figure 17: Second version on Ubuntu's dark mode

After a discussion upon the aesthetics and functionality of where the playlist should be put, we decided to use what you observe in the second sample. For this defense we're creating a playlist of songs even for a single song because it's more manageable which is also the reason why we picked the second sample. Furthermore, it is noticeable that there is a background color change in these three samples which comes from the theme that user uses. On Ubuntu, if one uses a dark theme, then the our application would look like the second sample, otherwise the first or second one is the default.

6.6 The last defense

For this last defense, all the features that were supposed to be implemented and linked to the GUI are done. Which lead us to add small but good details to the interface such as multiple dialog widgets to handle problems and guide the user when it is used in the wrong way.

6.6.1 Encoder

The encoding process which have been done since the second defense is finally included in the GUI! It retrieves the information of the actual song and wait for the input of the user. Once the changes have been saved, only information which are not empty or different than the actual information will be taken in account. Once the changes are done, a new song file with those changed information will be generated in same path.

Figure 18: A slice to show the edit information dialog box.

6.6.2 Converter

To use the converter, one should choose the song to compress, then a new dialog will pop up. One will then enter a name for the conversion, which will also create a new file with the compressed format. To have more information on the compression, please read the section 5 (Creating a new file format).

6.6.3 Dynamic logo for file formats

For bring more dynamism and content for the view of the user, the GUI displays by default a logo for all type of music. But it changes when the chosen song is of type MP3 or WAV.

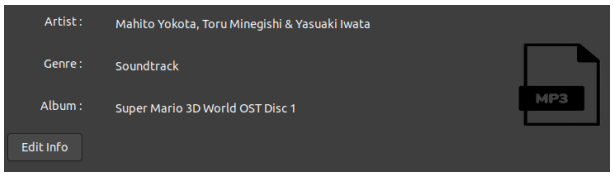


Figure 19: A view of a chosen MP3 song.

Figure 20: A view of a chosen WAV song.

Figure 21: A view of the default logo.

6.6.4 About

Among new features that are added, we have a dynamic about box that makes the interface looks more complete with the information on the project and the members!
As one can see below, that's what it looks like.

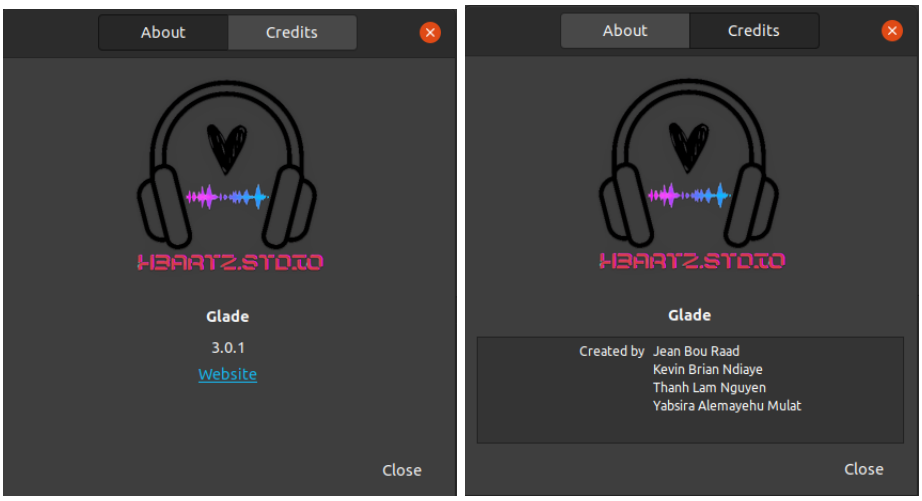
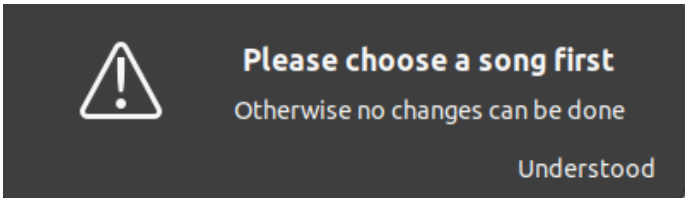


Figure 22: The dynamic About Us.

6.6.5 Warning Dialogs

As said in the introduction, there are multiples warning or message dialogs that have been created in order to tell the user what is happening and how the user interface should be used for a specific purpose. Let's see an example bellow.



This warning pops up whenever the user wants to edit the information of the playing song but there is no song chosen in the main widget.

6.6.6 File chooser extensions

In order to avoid errors because the user chose the wrong type of file in the file chooser, we added extensions to it, and now it will only display folders and only songs of type MP3, WAV and FLAC will be displayed. This feature will also make the user's experience better because the file chooser will look less messy.

6.7 Conclusion

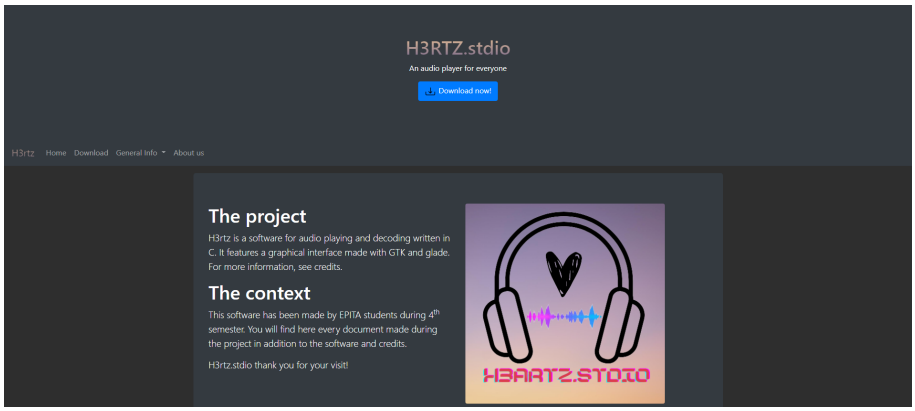
The end of the project is finally reached and H3rtz.stdio is proud to present the completed version of the user interface!

Chapter 7

Website

With the project, we must provide a functional website to showcase the evolution and the features. Its address is:

https://areas0.github.io/h3rtz/



A screenshot of our home page

7.1 Design

Our website uses a template created for the S2 project by Jean[1]. With some improvements plus some editing, the website is already functional. It features:
— A classic homepage
— A download page with some documents
— A task & issues page
— A page for the credits
— An "about us" page

[1] https://areas0.github.io/website/

Our website fits all the requirements for the last presentation and offers a diverse and rich experience with the latest improvements done for this defense.

The homepage features a small news section, a menu with some additional and useful links.

The download page is a place for all documents, executable made during the semester.

The task and issues page offer some details concerning how we have progressed overtime on the project. It will include some information on the bugs we have fixed.

The "about us" page includes a description of the group and all group members.

The main framework used is Bootstrap[2]. It offers dynamic layering to adapt the website to all sizes of screens.

7.2 Hosting

The website is hosted on GitHub Pages[3]. It offers reliable and easy-to-use service to host a simple website that requires no dynamic changes. You have probably guessed it, but the website must be in a GitHub repository to be hosted and updated. Updates are done by following the commits on the repository's master branch.

[2] https://getbootstrap.com/
[3] https://pages.github.com/

Chapter 8

Book of specifications: follow-up

8.1 Assignment tabular

We will assign a letter to every member to make our schedule more readable

1. KB : Kevin-Brian
2. J : Jean
3. L : Lam
4. Y : Yabs

Tasks	KB	J	L	Y
Multi-threading	×	×	×	×
Audio formats	×	×		
File Compression		×		×
GUI			×	×
Website		×	+	

Cross : (×) Person in charge and Plus : (+) Assistant.
As you can see multi-threading will involve everybody.

8.2 Progression

We needed to consider how we were going to handle our time based on the three main deadlines :

1. The 1st presentation (March 29th - 2nd April, 2021)
2. The 2nd presentation (3rd - 7th May, 2021)
3. The 3rd presentation (14th - 18th June, 2021)

We made a tabular to make it easy to read. The 1st tabular below is the progression goals that we set from the beginning of the project :

Tasks	1st	2nd	3rd
Multi-threading	30	70	100
Audio encode - WAV	30	100	100
Audio decode - WAV	80	100	100
Audio encode - MP3	0	60	100
Audio decode - MP3	30	80	100
User Interface	40	70	100
Conversion Support	0	40	100
Website	100	100	100

And this is our actual progression on the project :

Tasks	1st	2nd	3rd
Multi-threading	75	90	100
Audio encode - WAV	70	100	100
Audio decode - WAV	80	100	100
Audio encode - MP3	0	35	35
Audio decode - MP3 (GST)	75	90	100
Audio encode - H3Z (NEW)	0	0	100
Audio decode - H3Z (NEW)	0	0	100
User Interface	50	70	100
Conversion Support	0	40	100
Website	100	100	100

We changed the book of specifications once for the first defense to remove parts of the projects too complicated to realize in the frame of time that we had. Moreover, due to the level of complexity of the MP3 encoding, we were forced to abandon also that part of the project. We replaced it with a lossless file format created by the group. For more details, please see the previous chapters.

Chapter 9

Conclusion

This report concludes our one semester wide journey. It had its ups and downs with great successes and some failures with the MP3 file format.
With the failures, we learned that we could be creative and create a file format using existing norms. With the successes we had, we built a solid base of knowledge which will be handy next year.

We built a GUI with a complete set of features. It is followed by a full backend that provides audio playback via PulseAudio & algorithms for file compression. These features are well polished and perform well thanks to the wide use of deferred callbacks and multi-threading. We are proud of the global result and we had a good time working on this project.

Thank you.