H3rtz.stdio

Report #1

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

Contents

1 Introduction

In this first report, we will present the main achievements of our team during this first period. A lot has been done and overall we have made good progress on both audio decoding and the user interface.

Now that we have better knowledge about audio decoding, we are more aware of the difficulties that are to come. Hence, we delivered in addition to this report a new book of specification.

The first part of our report will be about the WAV file format with its decoding and encoding. Our team has achieved a very complete header parser and a prepared a raw to wav format encoder. These features are the basis used to decode other file formats such as MP3, AAC, FLAC etc...

The second one will bring details on the implemented of the audio player via PulseAudio. The audio back-end is almost fully implemented. This part will detail the steps and concepts around the implementation of such a player.

Finally, the progress on user interface will be shown. The part showcases the integration between the audio back-end and GTK, and the work done on design with Glade.

2 WAV decoding

First thing off, the wave format is old, and so it has documentation. As a reminder, its main characteristics are:

1. Uncompressed: this format holds most of the time PCM (pulse-code modulation). It is given by most software to the audio driver to create sounds. One could say that it is as low level for audio as C is for programming languages.

2. Wide support: many operating systems support it. However, some can have equivalents (e.g.: Mac OS).

3. Consistent: the format has not evolved much over the years. Hence, it was easier to determine all cases for the header.

With all the above, one must note that files are divided into blocks of data. The data inside a file is not only PCM data. They contain most of the information about the file itself. It can be the bitrate, the song's title, the type of audio, etc...

2.1 Parsing the headers: global process

All audio formats contain a header. It helps the software know how to decompress the data correctly (if the audio is compressed), give the audio driver-specific information such as the bitrate, the number of bits per sample. There are multiple types of blocks in a WAV file: RIFF, FMT (3 variations), FACT, DATA, LIST. Each of them must be parsed separately and have its specifications.

While reading the data mapped in memory using the function mmap, we need to identify each block. How do we do that? According to the standard, all chunks begin with their identifier: those who were listed above. However, they are only four characters without the ending (NULL) character. So, to correctly detect the names, we used their big-endian value as four characters.

For instance, in big-endian, RIFF can be written as four single bytes with their ASCII code: R: 0x52 I: 0x49, F: 0x46 (two times). Together, they form 0x52494646. For instance, "DATA" becomes 0x64617461. So, we need to compare those values to the ones currently being read by the program. Using some bit-shifting operation, we can easily convert bytes to 32 bits integers. The code is as follows:

```
// Fast 4 bytes to int conversion (merging) BIG ENDIAN
// @param data: must be an at least 4 bytes long array
__int32_t intConversionBE(unsigned char *data)
{
    return (data[0] << 24) | (data[1] << 16) | (data[2] << 8) | (
    data[3]);
}
```

Figure 1: The integer conversion function

Once identified, a block always contains after its identifier a size (regular integer). Finally, after that, it varies from header to header. Hence, the next parts will detain each block, their specification, and how they are parsed by the program. Even though this does not matter at such a low scale, this method is way faster than traditional comparison between strings with *strcmp*. Indeed, four bit-shifting operations and one comparison are needed.

Our wrapper for the complete header is as follows:

```
// 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;
```

2.2 Parsing the RIFF

Below, you can find an example of a RIFF block.

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

figure 2: Typical riff structure

As usual, the four first bytes contain the string "RIFF" and then the next four bytes hold the file size. Indeed, the RIFF is considered as a master block of the file. After the size, the file has another identifier: the main format of the file: wave. If it is not wave, then the program will **abort** the process because the data is not a wave file.

2.3 Parsing the DATA block

Here, you can find the typical structure of a DATA chunk:

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

Figure 3: Typical data structure

From now on, we will **not** remind you the details about the two first parts of each block being the identifier and its size.

This block holds the audio data, the one that must be played by the program. Hence, this block is larger than the others.
The attribute chunk simply points to the beginning of the audio data. It is already allocated in the memory thanks to the initial mapping.

2.4 Parsing the FMT

This block is named FMT as a shortcut for FORMAT. Hence, one can deduce easily that it contains information about the audio format. In terms of importance, this block ranks **high**. There are **three** variants of this header (classic, float, extensible). We will provide more details concerning them later in this part. Below, you can find the classic version of the format block.

```
typedef struct fmt
{
    //format block id 4B
    char fmt[4];
    // classic 16b
    int blocSize;
    //2B see enum compression_codes
    unsigned short AudioFormat;
    short channel; //2B

    int sampling_freq; // 4B
    //important 4B //number of bytes to read per sec
    int bitrate;
    //2B NbrChannels * BitsPerSample/8
    short blocrate;
    //2B //8-16-24-32-64 bits
    short samplerate;
} fmt;
```

Figure 4: Basic format (FMT) structure

The first attribute is an identifier (*AudioFormat*). Those help the program determine the kind of audio data inside. Technically, the wave standard supports many file formats but most of them are rarely used. Below, you can find the most used in an enumeration to identify them:

```
//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, //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 5: List of formats with their identifier as integers

Then there is the attribute channel. It indicates if the file has stereo, mono, or multi-channel audio data. Again, usually the most common are stereo and mono.
The sampling frequency (44100 Hz, 48000 Hz, 96000 Hz for instance), the bitrate (varies from file to file), and the number of bits per sample, are also available in this block. Concerning the variations, they add data to the structure without modifying the existing attributes (order and size). The IEEE float PCM variation adds an attribute to get the offset to read the file correctly.

The second extension, the wave extensible format, can be considered as a sub-block.

```
// 40 bytes bloc WAVE_EXTENSIBLE
typedef struct fmt_extensible
{
    // same as figure 4...
    short extension_size;
    short valid_bytes_ps;
    int channel_mask;
    char sub_format[16]; //contains the new format
} fmt_extensible;
```

Figure 6: Typical extension of the format structure (everything above remains the same)

When this extension is present in a file, the audio identifier is replaced with an indicator to tell the software to look for additional information (see WaveFormatExtensible in the enumeration compression codes above). There is one

major add-on to this format: sub-formats codes and channels masks. The last one provides a repartition of speakers in space when playing multi-channel audio (for instance 5.1/7.1 surround audio). For the sake of simplicity, we decided to ignore this data when playing audio, as we aim to play mono and stereo data correctly.
Concerning the sub-format identifier, it is a 16 bytes (128 bits) identifier. It is the equivalent of compression codes but in a wider range.
However, this led to an unexpected challenge: handling 128-bits integers.
With C99, 128 bits integers are included as a standard type. However, compilers such as GCC or CLANG **cannot** handle 128-bits integers constants nor have enumerations with this kind of indexing. Moreover, support varies from platform to platform, hence the need for this easiest solution. To bypass GCC's limitations with enumeration, we used constants. However, as stated, GCC cannot compile directly 128-bits constants. So, instead, we used bit shifting to get it right. Indeed, GCC has no problem evaluating 128-bits expressions **while** compiling. We split the number into two longs and shifted the first part 64-bits to the left. Below, you can find an example:

```
1 // compiler constants cannot be larger than 64 bits so bit shifting
      must be used
2 const __uint128_t PCM_CODE = ((__uint128_t) 0x0100000000001000 <<
    64) | 0x800000aa00389b71;
```

Figure 7: example of a 128-bits integer constant definition to trick GCC

With some modifications to the previous method, the conversion from the array of characters was easily done to compare with those constants. As you may have noticed, the format block is held in the main structure thanks to a union. We provide a link to explain what a union is, just in case: https://www.wikiwand.com/en/Union_type.
It is convenient for our use case because all the variants of our format block have their common attributes in the same order and occupy the same space in memory. So, whatever the type of block, accessing the attribute bitrate via the fmt_float, fmt (classic), fmt_extensible, will not change anything. And when we need to know which kind of format block we have, the main structure contains an enumeration to indicate that.

2.5 Parsing the FACT block

This block is optional. It is present most of the time when the audio is PCM float data. It contains little information, which is not essential to our project, so we will not provide details concerning their roles.

2.6 Parsing the LIST block

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

```
// Contains various info
// linked list with sentinel structure
typedef struct info
{
    char infoId[4];
    unsigned int size;
    char *data;
    struct info *next;
} info;

typedef struct list
{
    char list[4];
    int chunk_size;
    unsigned char *data;
    struct info *infos;
} list;
```

Figure 8: Definition of the INFO and LIST chunks

Each node contains an identifier. It says what the data is about. Below, you can find an enumeration with the most useful identifiers:

```
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
    ISFT = 0x49534654, // software
    ISRC = 0x49535243, // source
    IPRD = 0x49505244, // Product
    IPRT = 0x49505254, // track id
} infoIds;
```

Figure 9: Enumeration of the information identifiers with their long value

Then there is a pointer to the key for the node.
Of course, we implemented methods to allocate and free the linked list.

2.7 Conclusion

Our current implementation of wav files is flexible and reliable. It offers a set of functionalities, which can be considered as very complete. It has been tested on various sets of files and has shown great performance with a parsing done in less than 2 ms on average.

3 WAV encoding

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 or later the MP3 file.

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.

3.1 Writing the raw file

Writing a RAW file from a WAV file boils down to two things :

1. Figuring out the size of the header
2. Writing a proper data structure

Because the size of the header depends on how much information it holds, we need to figure out its size and use lseek(2).

The entire problem now becomes how can we figure out the exact size to avoid skipping critical data and creating a partial signal for the new WAV file which won't be readable.

Conventions can change depending on the amount of data written into the header.

Simple files will have different ways of writing data compared to files containing long headers.

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

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

1. 3 null bytes will follow the size of the current chunk

2. If the size of the chunk is odd, a null bytes is added to balance everything out.

3. And the rest contains conventions for what to name each chunk (artist, genre and etc...)

The function:

```
size_t size_header(struct wav *h);
```

We keep the data section instead of the raw signal for extra information once we open the file.

3.2 Writing the new header

Using the terminal and next time 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 (old_file_2.wav) and a new header (either from the old file or the user).

```
void write_header(int fd, struct wav *header);
```

3.3 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. The raw data is obviously written in hexadecimal so we found a way around it.

```
1  //writes n null bytes for spacing
2  void write_space(int fd, size_t n)
3  {
4      for (size_t i = 0; i < n; i++)
5      {
6          unsigned char hex = 0x0;
7          if (write(fd, &hex, 1) < 0)
8          {
9              errx(1, "encode: write spacing failed");
10         }
11     }
12 }
```

3.4 Conclusion

The WAV encoding was not challenging algorithmically but inconsistencies in the header proved to be difficult and eerie.

So far, we used 14 samples exploring most of what WAV files have to offer. The WAV encoding program works with 10 out of 14 for now.

4 Audio player: PulseAudio

To play raw PCM signals, we decided to use PulseAudio. Pulseaudio is software available on every major distribution of Linux except one. An API is also available to communicate with the software.
Hence, it was the perfect candidate for our project. There are other options, for instance, with ALSA because PulseAudio is built on top of it. However, its overall complexity was considered too hard.
We preferred to focus on other important parts of our project (audio decompression). We will go into the details of the implementation, but one of the advantages of PulseAudio is that it provides a multi-threaded and asynchronous API. These features are mandatory to run a GTK interface in parallel.

All the functions mentioned are listed and explained in the following documentation: https://freedesktop.org/software/pulseaudio/doxygen/index.html

4.1 Principle of implementation

PulseAudio's API provides a threaded mainloop. It relies on event polling to send and receive signals when tasks are being run asynchronously. However, these principles are new to us. There are multiple obligatory steps to get a PulseAudio player running and playing audio correctly. First, initialize a mainloop. It is the main piece of our program. The loop itself holds an API that can be used for some particular operations.
The program needs to connect to the PulseAudio server running on the user's computer.
Then, it must create a stream to play audio smoothly.

Finally, once initialized, we have to pass the audio data to the streamer.
Of course, as for any asynchronous multi-threaded library with events, PulseAudio uses callbacks for many things. One of these callbacks must be a function to write data to a buffer provided by PulseAudio. Deferred callbacks do not block our program. Hence, our interface can run while another thread is writing to a buffer provided by the library.
This little paragraph represents only the **outline**. Many features are omitted and will be detailed in the later parts.

4.2 Implementation

4.2.1 Structures

We already explained that PulseAudio works with many objects. Callbacks and other functions need some of them to work. Hence, some structure was needed. First, we have the main structure that contains an entry point to some file's data and PulseAudio's objects:

```
typedef struct pa_player
{
    wav_player *player;
    pa_objects *pulseAudio;
    pa_info *info;
    state pa_state; //enumeration
    fileType type; // enumeration
    pa_time *utility;
} pa_player;
```

Figure 10: Structure definition of a PulseAudio player

For this part, we will ignore the attributes player and *pa_state*.
The type *pa_objects* is our wrapper for all PulseAudio's objects:

```
typedef struct pa_objects
{
    pa_context *context;
    pa_threaded_mainloop *loop;
    pa_stream *stream;
    pa_mainloop_api *api;
    char *sink;
    pa_usec_t *latency;
    pa_server_info *server;
} pa_objects;
```

Figure 11: Definition of the PulseAudio objects structure

We give to callbacks and functions, linked to that part of the project most of the time, a *pa_player*.
It is now the time for an introduction to the player states. We created an enumeration to determine the current state of our PulseAudio player. For instance, there are states for the paused, playing, drained player. They are useful, especially for deferred callbacks. Indeed, they help prevent unexpected behavior such as trying to drain a stream already empty. The enumeration is as follows:

```
typedef enum state
{
    READY,
    PLAYING,
    PAUSED,
    FINISHED,
    DRAINED,
    TERMINATED,
} state;
```

Figure 12: enumeration containing the different states of our player

4.2.2 Initialization

The two following functions are used by the program to initialize a *mainloop* and then set up a stream:

```
int init_player(pa_player *player);
int prepareStream(pa_player *player);
```

Figure 13: two prototypes related to initialization

They both return integers to have error handling with the interface. The first function prepares a mainloop and connects it to a context. The context is automatically linked by PulseAudio to an audio server. Those steps are immutable, to be known, and examples in the documentation helped us get started.
These elements are then used by the second function to create a stream. 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* and *pa_stream_connect_playback*, we can create a stream and connect it to the device (sink).

Our first implementation was functional but not ideal. Indeed, PulseAudio evolves, new features are added. PulseAudio recently introduced a new initializer for streams. It is the function *pa_stream_new_extended*. PulseAudio's documentation is already not very complete and not easy to understand. With that, it got worse: no mention of these functions in the main documentation files. The implementation was done only with the help provided in the headers of the library. But why did we decide to implement it? Because it allows us to provide more data to the player concerning the file being played.

We understood how this new function worked after many hours of digging into PulseAudio's documentation. The initializer requires a *pa_format_info* object. It merges the *pa_sample_spec* with the *pa_channel_map* but can contain more data. A method in the library can set the *format_info* with the two previous objects: *pa_format_info_from_sample_spec*. We can then add some more information about the file being played using the info chunk from our wav file.

4.2.3 Writing the audio data

This part is critical because it must be fast and reliable. The callback function is as follows:

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

Figure 14: declaration of PulseAudio write callback function

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*.

```c
typedef struct wav_player
{
    wav *info; // the header
    file *track; // the mapping of the file
    unsigned char *data; // pointer to the beginning of the
    data
    size_t offset; // reading offset
    double time; // current timestamp
} wav_player;
```

Figure 15: Declaration of the *wav player* structure

4. Write it to the real buffer with *pa_stream_write*.

5. Update the offset for the file and the number of bytes written.

4.2.4 Terminating & draining a stream

As for memory allocation, cleaning a stream after use is necessary.
This operation is not trivial. Indeed, cutting a stream too early can discard some audio data that needs to be played. For instance, after writing the last few bytes, PulseAudio keeps around two seconds of data in its buffer. Cutting as soon as the last byte is written would remove these two seconds.
To avoid such a thing, PulseAudio provides a draining function. A call to this procedure returns a *pa_operation* object. This kind of object means that the operation is run asynchronously, and will call a deferred callback function. To verify that this operation is complete, we used a classical signal structure that follows the steps below:

```c
// Makes sure that we played all the samples before disconnecting
    the stream
void drainStream(pa_player *player)
{
    pa_objects *pa = player->pulseAudio;
    pa_threaded_mainloop *loop = pa->loop;
    pa_stream *stream = pa->stream;
    // locks the mainloop to avoid errors
    pa_threaded_mainloop_lock(loop);
    // starts a draining operation, it is asynchronous
    pa_operation *op = pa_stream_drain(stream, &callbackDrain, loop
    );
    // we wait for it to be done
    while (pa_operation_get_state(op) != PA_OPERATION_DONE)
        pa_threaded_mainloop_wait(loop);
    player->pa_state = DRAINED; // the operation is done
    pa_operation_unref(op);
    // we can unlock and continue business as usual
    pa_threaded_mainloop_unlock(loop);
}
```

Figure 16: example of a typical draining process with multi-threading and deferred callbacks

If you have already done some multithreading in C, this should look similar to the structure with pthreads. We lock first the main thread, then wait for a signal from the deferred callback. Finally, we can disconnect the stream and unlock our thread. When we want to switch tracks quickly, another procedure must be used. It works as follows:

1. Suspend the current stream - sink with *pa_context_suspend_sink_by_name* (asynchronous)
2. Disconnect then the stream with *pa_stream_disconnect*
3. Dereferencing the stream with *pa_stream_unref*

Then, if we want to play another file, we must create a new stream. The *prepareStream* alone does the job.

4.2.5 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.

4.2.6 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. It is not yet available in the interface.
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:

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

Figure 17: 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 18: 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.

4.2.7 Forwarding/Rewinding

Forwarding or rewinding during playback is always a story about offsets. When we explained the write callback, we did not mention the fact that the *pa_stream_write* has several modes to write at different relative positions in the buffer.

```
/** Seek type for pa_stream_write(). */
typedef enum pa_seek_mode {
    PA_SEEK_RELATIVE = 0,
    /**< Seek relative to the write index. */

    PA_SEEK_ABSOLUTE = 1,
    /**< Seek relative to the start of the buffer queue. */

    PA_SEEK_RELATIVE_ON_READ = 2,
    /**< Seek relative to the read index. */

    PA_SEEK_RELATIVE_END = 3
    /**< Seek relative to the current end of the buffer queue. */
} pa_seek_mode_t;
```

Figure 19: Extract from PulseAudio's headers with the different types of writes

We usually use the relative mode. However, we need to use an absolute position because we need to rewrite the buffer completely. Via a modified callback function called by the interface, the program is allowed to go to another timestamp quickly. It is not currently implemented in the interface, but it will be for the second defense.

4.2.8 MP3 and other formats playback

Due to high complexity of decompression, especially with the number of algorithms involved, we decided to avoid doing decompression for the moment. This led to the implementation of a fast audio decoder for compressed formats. We might implement a solution for audio decoding by hand later.

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 currently 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 a little slow. However, this was the best solution to build a complete audio player.

4.2.9 Testing & Struggles

With PulseAudio came a lot of challenges and difficulties. The use of libraries combined with multi-threading can make debugging complicated or impossible. Adding on top of that the implementation of our interface with GTK, and you have got hieroglyphs to decode. As a result, it led to a very isolated construction of our audio back-end to ensure its stability with some tools for testing.
For instance, we have implemented a shell player. It allows us to easily test many features such as the system of timestamps with latency, the parsing of information from the file.

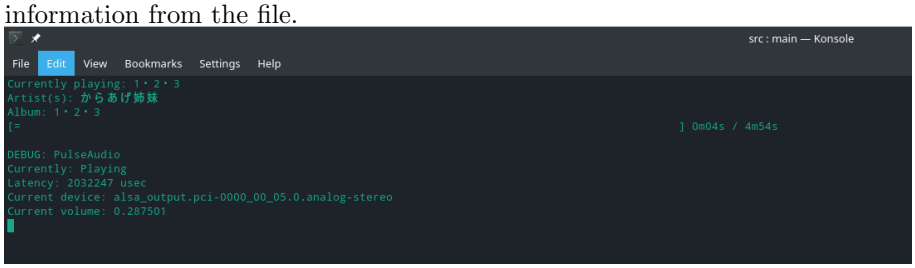


Figure 20: An example of debugging session on shell (Manjaro, Konsole)

Of course, it would not be fun if it was not multi-threaded and built using semaphores to ensure light CPU usage.
Finally, using the known tools such as GDB, Asan, Valgrind, we were able to trace all the bugs we were able to produce. After all, debugging is like investigating a murder knowing that you're the culprit.

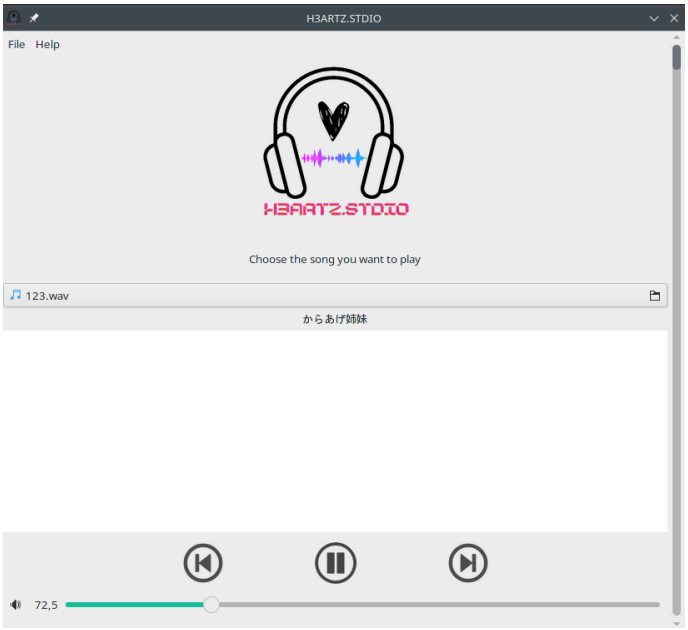
4.3 Conclusion

Our current implementation is almost complete. It features the most common properties that one could find on an audio player. It still has to be extended, improved, polished with more features. For instance, we kill and then start a new stream for each file. We will try to implement a dynamic streamer for better performance.
Working so far with PulseAudio was a bit painful due to the lack of decent examples and a lack of knowledge.

5 GUI: User Interface

5.1 Glade

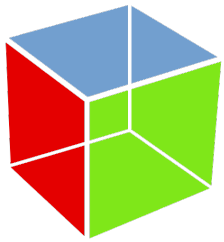
Glade is a RAD tool to enable quick & easy development of user interfaces for the GTK toolkit and the GNOME desktop environment.
The user interfaces designed in Glade are saved as XML, and by using the GtkBuilder GTK object these can be loaded by applications dynamically as needed.
For our project, we 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/

5.2 GTK



GTK is a free and open-source cross-platform widget toolkit for creating graphical user interfaces.

5.3 Components of the Interface

5.3.1 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  }
```

5.3.2 Play and Pause button

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

5.3.3 Progress Bar

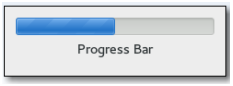
As one guess from the name, the progress bar helps us to show the progress of a playing audio. It gives information about the time or duration of the audio. To implement this, there are two methods that we used. The methods are : -

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

GtkProgress Bar

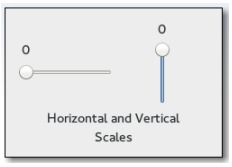
The GtkProgress Bar is a widget which indicates progress visually. It shows the progress of some process using percentages. For our project, we implemented the GtkProgress Bar but for the sake of dynamics the interface, we used the

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.



The implementation is as follows: -

```
1  int slider(void *userdata)
2  {
3    gtk_player *player = userdata;
4    pa_player *pulseAudio = player->player;
5    // Total duration computation: nb bytes in file / bitrate
6    if (player->player->pa_state >= DRAINED)
7    {
8      gdouble max = gtk_adjustment_get_upper(player->ui.adjustment);
9      // 100% completion for interruption
10     gtk_adjustment_set_value(player->ui.adjustment, max);
11     // disables callback
12     g_source_remove(player->ui.ID);
13     GtkWidget *image = gtk_image_new_from_file("./GUI/callbacks/
       icons/play.png");
14     gtk_button_set_image (GTK_BUTTON(player->ui.play_pause), image)
       ;
15     return FALSE;
16   }
17   double duration = (double)pulseAudio->player->info->data->
       data_bytes /
18                     (double)pulseAudio->player->info->fmt->bitrate;
19   gtk_adjustment_set_upper (player->ui.adjustment,duration);
20   // updates latency from pulseAudio
21   updateLatency(pulseAudio);
```

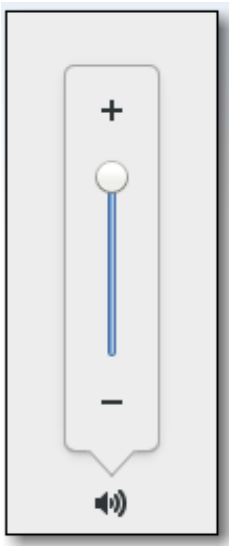
```
22    // computation zone
23    double timing = pulseAudio->player->time;
24    // converts to microseconds
25    timing *= power;
26    // removes latency
27    double wLatency = timing - (double)*(pulseAudio->pulseAudio->
        latency);
28    //ratio = ((wLatency / power)* 100) / duration;
29    double current = wLatency/power;
30    player->player->utility->current = current;
31    // update the progress bar
32    gtk_adjustment_set_value (player->ui.adjustment, current);
33    gtk_scale_set_draw_value(player->ui.slider, TRUE);
34    return TRUE;
35 }
```

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

5.3.4 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: -

```
void cvolume(GtkWidget *widget __attribute__((unused)), gpointer
    userdata)
{
    gtk_player* player = userdata;
    if (player->player->pa_state >= DRAINED)
        return;
    gdouble value = gtk_scale_button_get_value(GTK_SCALE_BUTTON(
    player->ui.volume));
    player->player->info->volume = value;
    setVolume(player->player);
    return;
}
```

5.3.5 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:

```
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

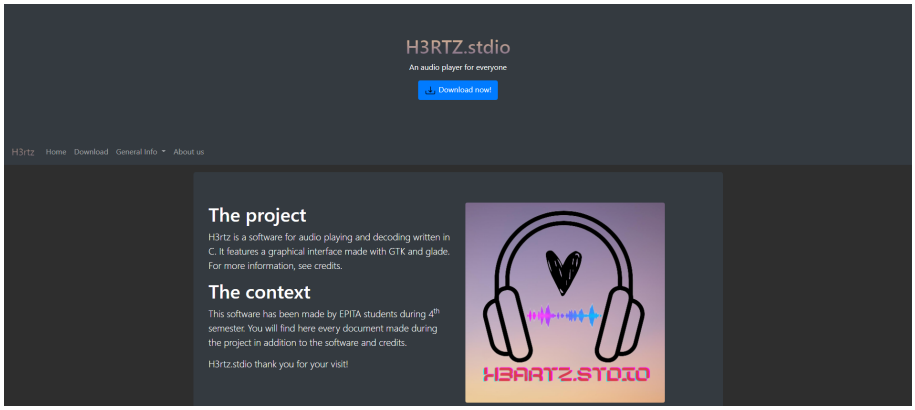
5.4 Conclusion

To conclude, the objectives for this first defense about the interface are reached. Our team is working on the playlist implementation and a better looking interface for the next defense.

6 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

6.1 Design

Our website uses a template created for the S2 project by Jean[0]. 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

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.

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

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

The main framework used is Bootstrap[1]. It offers dynamic layering to adapt the website to all sizes of screens.

6.2 Hosting

The website is hosted on GitHub Pages[2]. 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.

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

7 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	×	×		
GUI			×	×
Website		×	+	

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

8 Progression

Now, we need to consider how we are 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 bellow 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	60	100
Audio decode - MP3 (GST)	75	90	100
User Interface	50	70	100
Conversion Support	0	40	100
Website	100	100	100

9 Conclusion

Our team has achieved good progress for this first defense. With an already functional audio player and its interface, the project has already passed some major milestones.

For the next defense, there will be a new set of skills involved with the beginning of algorithmic work for the encoding of the MP3 format. We will also continue improving the interface, the audio back-end to provide a more complete set of features.

Finally, except for the changes on some tasks (see the book of specification), we are currently on schedule and confident on the fact that we will be able to finish our tasks in time.

Thank you.