UNIVERSITY OF WEST LONDON

TC70031E – Audio Programming 1

Assessment 2

Harriet Rena Cerys Drury

Student ID: 21474551

28th January 2022

(Semester 1 2021.10 – 2022.2)

Abstract

This report highlights the research and implementation of the audio file format WAV. Research is focused on the history, usage, and advantages to this file format, alongside comparisons to other audio file formats such as the MP3.

Additionally, this report is used to comment on given code to evaluate the input of a WAV file in C++. This commenting is done with snippets of code analysed, along with flowcharts of sections to fully realise the code. The code given is written all in the main() function of the C++ language, therefore additional programming is done in the programming section to change this code to be object orientated.

The third portion of the report focuses on the programming task of assessment 2. This task is to perform a sample rate change and mute on a mono WAV audio file. The result of this program is an object orientated approach with flowcharts provided for the sample rate change and mute. Results are discussed with the visualisation of the waveforms in MATLAB. It is concluded that the sample rate is successfully changed in both the meta data and audio data of the WAV audio file. Future work is discussed to include more user input features and a header file to place constants in.

Table of Contents

Table of Figures

<u>Table of Tables</u>

<u>Table of Equations</u>

1. Introduction

Overview

This report is created to collate and discuss research and programming surrounding the given task of working with the WAV audio file. This file type uses a resource interchange file format (RIFF) type to hold both the audio data and metadata to recreate the audio in programs. Research into this style of working is first discussed.

The programming given in the lecture Week 11 to do simple audio file input/ output is discussed and commented on. It is also researched and placed into context of audio processing.

Finally, the final portion of this report works with manipulating the WAV format. This includes changing sampling rates and muting a portion of the audio file.

Focus and Structure of the Report

The main focus of this report is to highlight the research and implementation of WAV file formatting. The manipulation of the WAV file is also inputted, as well as using figures and flowcharts to support points with visual feedback.

The methodology is split into three portions; the research, the given code and the programming assignment. Each portion works with separate parts of the aim of understanding the WAV audio file. The results of the programming assignment is discussed and supported with additional programming in Matrix Laboratory (MATLAB).

2. Research

Introduction

For the first task, research is undertaken to utilise, understand, describe, and analyse the format of a WAV file. This includes researching the chunking of formatted data (RIFF formatting), endian formatting and hexadecimal placements.

The WAV file is an audio format standard outlined by IBM and Microsoft in 1991 (Microsoft and IBM, 1991). The Macintosh computer uses similar AIFF formatting to produce lossless audio bitstreams (Dempsey, 2020).

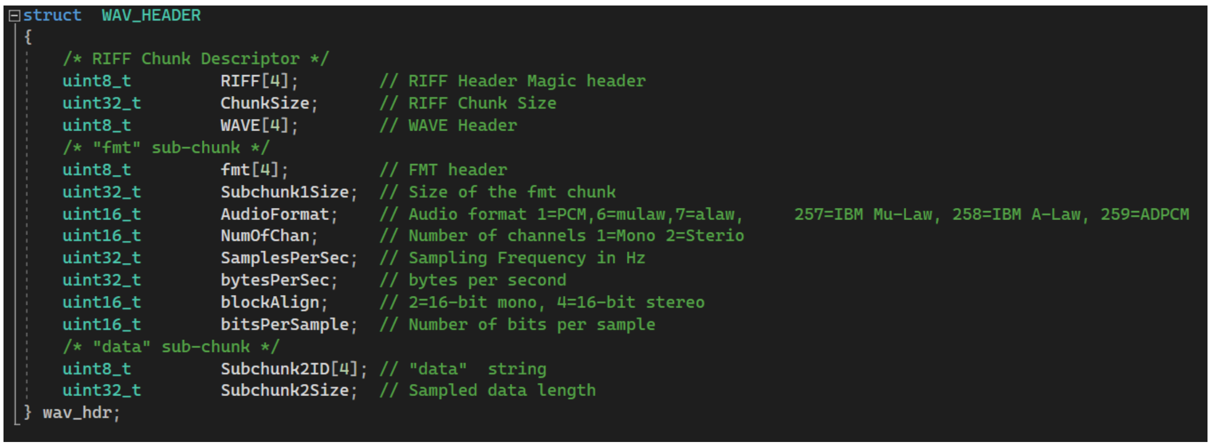
Research

The WAV file follows the resource interchange file format (RIFF) as set out by IBM and Microsoft (Microsoft and IBM, 1991). A RIFF file is composed of multiple discrete sections of data called chunks (Pohlmann, pp732 – 734). Within the multimedia WAV file format, the chunks are split into 3 sections, the RIFF identification formatting, the formatting of the audio and the data, shown in table 1 below.

CHUNK	UTILITY	BYTE SIZE
RIFF	Identifies Wav file & Length	12
FORMATTING	Sampling Frequency, Channels, etc	24
DATA	Chunk Length & Amplitude Sample Values	Dependent on Data

Table 1. The Chunks of the WAV File

In C++ this can be written in a struct data type to be called when required. This is shown in figure 2.1 below.



```
struct  WAV_HEADER
{
    /* RIFF Chunk Descriptor */
    uint8_t         RIFF[4];        // RIFF Header Magic header
    uint32_t        ChunkSize;      // RIFF Chunk Size
    uint8_t         WAVE[4];        // WAVE Header
    /* "fmt" sub-chunk */
    uint8_t         fmt[4];         // FMT header
    uint32_t        Subchunk1Size;  // Size of the fmt chunk
    uint16_t        AudioFormat;    // Audio format 1=PCM,6=mulaw,7=alaw,     257=IBM Mu-Law, 258=IBM A-Law, 259=ADPCM
    uint16_t        NumOfChan;      // Number of channels 1=Mono 2=Sterio
    uint32_t        SamplesPerSec;  // Sampling Frequency in Hz
    uint32_t        bytesPerSec;    // bytes per second
    uint16_t        blockAlign;     // 2=16-bit mono, 4=16-bit stereo
    uint16_t        bitsPerSample;  // Number of bits per sample
    /* "data" sub-chunk */
    uint8_t         Subchunk2ID[4]; // "data"  string
    uint32_t        Subchunk2Size;  // Sampled data length
} wav_hdr;
```

Figure 2.1 The Structure of a WAV file in C++

<u>RIFF Chunk</u>

This chunk allows for the identification of the RIFF formatting and WAV multimedia file choice.

Format Chunk

The bulk of the WAV file header information is stored in the format chunk. This section is identified with the header 'FMT '. Within the format chunk, information on the data kept for the audio is used to be able to recreate audio signals. This includes the format, the number of channels, sample rate and byte rate (Pohlmann, pp732 – 734). Audio format is important when working with audio files at the data packing for mono and stereo audio differs depending on the usage (Microsoft and IBM, 1991, pp58).

When the digital to analogue converter reads the WAV file, the format chunk allows for accurate recreation with the sampling rate and byte rate accurately describing each audio signal per second.

Various digital signal processing techniques are described within the format chunk so that software reading the WAV file can accurately parse data to the user.

Data Chunk

After the aforementioned metadata, the wave data is given in the data chunk. Due to the usage of a 32-bit unsigned integer to record the file size, the size of the WAV file is limited to 4GB.

Byte Formatting

Within the WAV file, chunks are formatted using little endian file formatting of hexadecimal numbers. Little endianness stores the least significant bit of a data series first, and the most significant bit last (Geeks for Geeks, 2021), in WAV files, 2 bytes are used to store data. In figure 2.2 below, LSB denotes the least significant bit and MSB signifies the most significant bit.

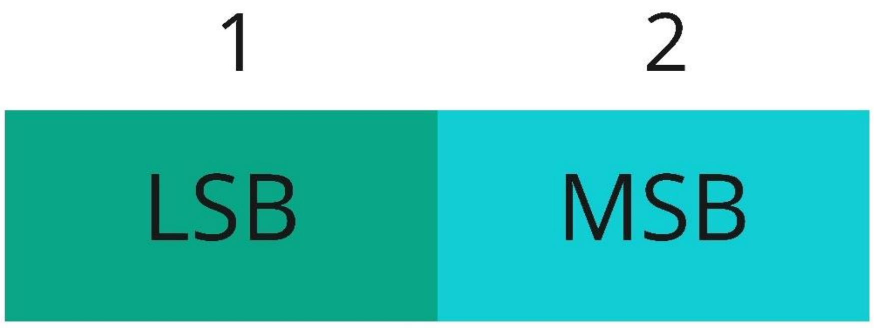


Figure 2.2 A diagram of Bytes Stored for WAVE formatting

Hexadecimal is a base 16 number system used to express values within the WAV file (Bird, 2017, pp33). The usage of hexadecimal allows for the expression of more data values with less bits. Within the WAV file, this is compensated for by switching the bytes around when decoding sections.

An advantage to working with the WAV file is that it is a lossless format. Each sample is declared within the file and does not use data minimising techniques (Pohlmann, 2010, pp51). The lack of data minimising can however cause issue to large data sizes due to high sample rates.

Comparison to the MP3 Audio File Format

A common comparison between WAV and MP3 files is drawn. The Mp3 is a coding format largely used due to the lossy data-compression to encode data and create approximations accurate enough to discard some data (Pohlmann, 2010, pp533). The reasoning behind comparison is for the choice of usage due to their different formatting standards. Table 2 highlights some differences between MP3 formatting and WAV formatting (Dempsey, 2020).

WAV File	MP3 File
Accurate, Lossless Format	Compresses Data with Little Overall Sound Quality Difference
High Dynamic Range	Sacrificed Dynamic Range and Quality
Large File Size	Small File Size

Table 2 Common Comparisons between WAV and MP3 Files

The implementation of WAV and MP3 files are observed to be generally for their different usages. The MP3 is largely used on portable devices with minimal data storage (Dempsey, 2020), whereas in production and audio editing setting, the WAV file is favoured.

Conclusion

To conclude this research section, the introduction of the WAV file by IBM and Microsoft allowed for easy lossless audio file transfer by using the RIFF file format. By using a Chunk based approach, easy access to audio meta data is observed, as well as the ability to manipulate the data stored. Further research breaks down the little endian formatting, which places the least significant bit in the two byte set at the front. Hexadecimal (base 16) numbering allows for the easy storage of more data in smaller data sizes. Advantages and disadvantages to the WAV file are briefly discussed. Finally, a comparison to the largely used MP3 is concluded to exist for other applications of audio files.

3. Commenting on Given Code

Introduction

The given code used within this exercise is from week 11 of the lecture series. Using this code allows for the integration of part three of the assignment, to build upon given code and create two new files with a changed sample rate and muted section respectively.

The given code used reads in a WAV file and parses the data chunk into normalised values within a vector. The header file data is also obtained so that the data size can be read and pointed to. Full code is given in Appendix 1. As previously discussed, a struct is declared to hold all the chunk information and is used to point to parts of the data, shown in figure 2.1. Figure 3.1 is the flowchart for the function of this code.

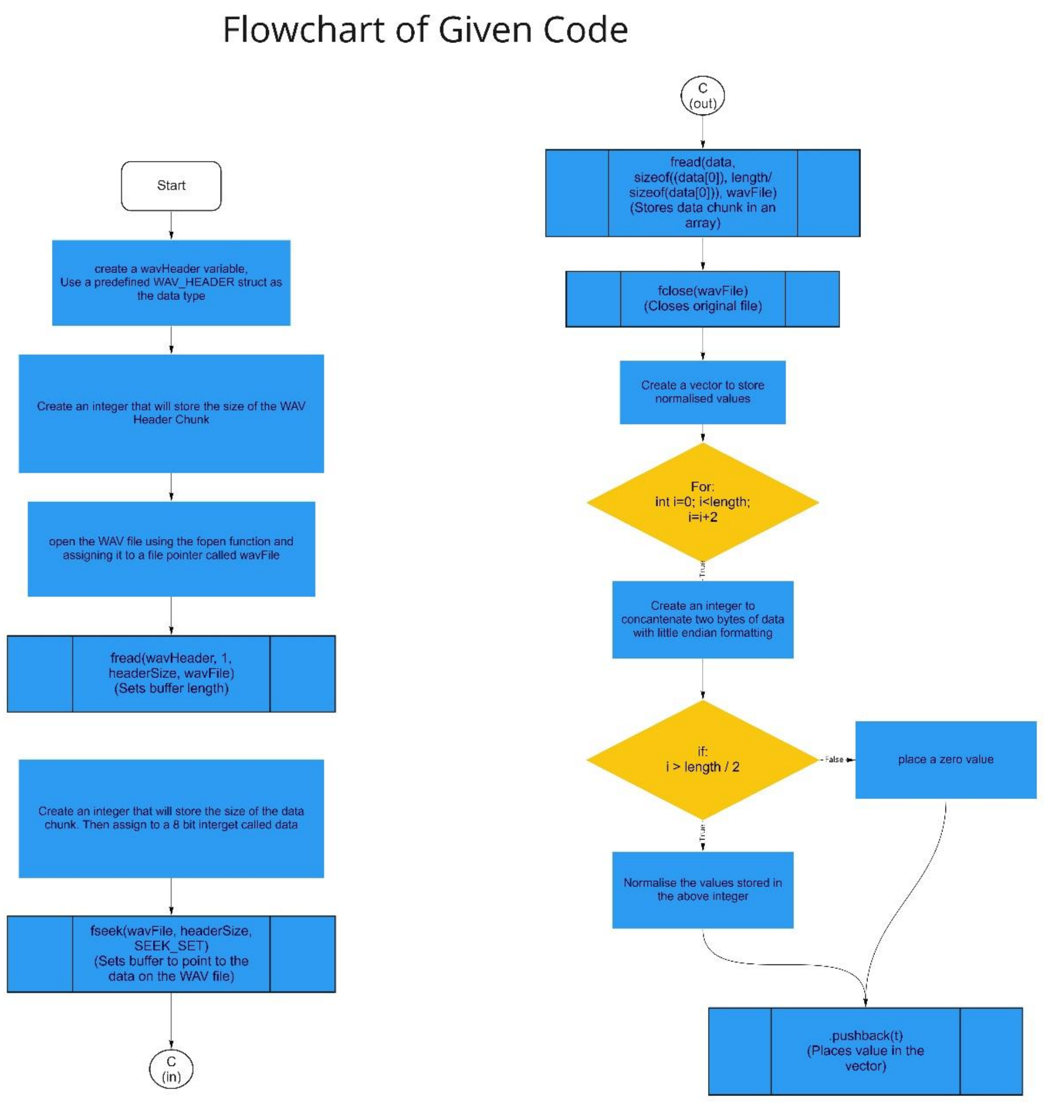
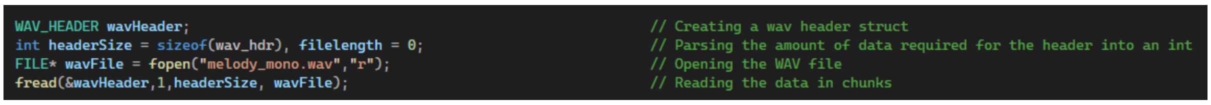

Flowchart of Given Code
Start
create a wavHeader variable, Use a predefined WAV_HEADER struct as the data type
Create an integer that will store the size of the WAV Header Chunk
open the WAV file using the fopen function and assigning it to a file pointer called wavFile
fread(wavHeader, 1, headerSize, wavFile) (Sets buffer length)
Create an integer that will store the size of the data chunk. Then assign to a 8 bit interget called data
fseek(wavFile, headerSize, SEEK_SET) (Sets buffer to point to the data on the WAV file)
C (in)
C (out)
fread(data, sizeof((data[0]), length/ sizeof(data[0])), wavFile) (Stores data chunk in an array)
fclose(wavFile) (Closes original file)
Create a vector to store normalised values
For: int i=0; i<length; i=i+2
True
Create an integer to concantenate two bytes of data with little endian formatting
if: i > length / 2
False
place a zero value
True
Normalise the values stored in the above integer
.pushback(t) (Places value in the vector)

Figure 3.1 A Flowchart for the Provided Code

Section 1 – Parsing the WAV header & Reading Data

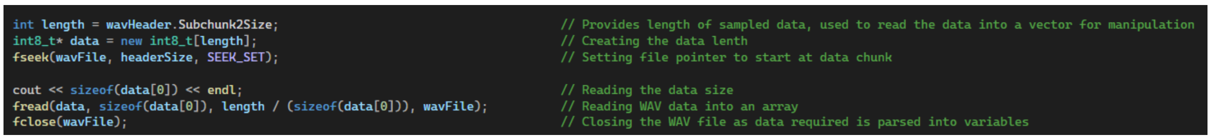
The first portion of this code highlights how the WAV file is placed into chunks with the header file as the reference for the placement. Figure 3.2 below shows this code.



```
WAV_HEADER wavHeader;                                                    // Creating a wav header struct
int headerSize = sizeof(wav_hdr), filelength = 0;                        // Parsing the amount of data required for the header into an int
FILE* wavFile = fopen("melody_mono.wav","r");                            // Opening the WAV file
fread(&wavHeader,1,headerSize, wavFile);                                 // Reading the data in chunks
```

Figure 3.2 The Header File Usage

From this code, we observe that the size of the header section is defined from the length of the generated struct WAV_HEADER. This in turn allows for the placement of the header file data to be pointed to by the fread() buffer. In doing this, the data portion of the WAV file is portioned out to be manipulated and easily pointed to by the code. The WAV file is also opened in ‘r’ mode which correlates to reading data from the file (Friedman and Koffman, 2007, pp398). The WAV file stays open for the parsing of data, shown in figure 3.3.



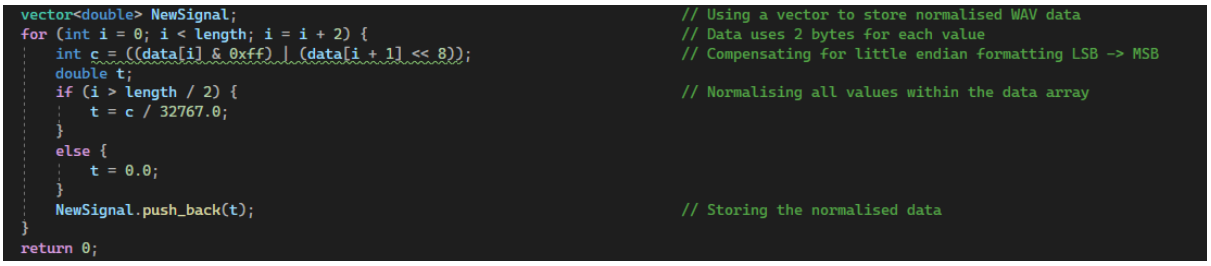
```
int length = wavHeader.Subchunk2Size;                                    // Provides length of sampled data, used to read the data into a vector for manipulation
int8_t* data = new int8_t[length];                                       // Creating the data lenth
fseek(wavFile, headerSize, SEEK_SET);                                    // Setting file pointer to start at data chunk

cout << sizeof(data[0]) << endl;                                         // Reading the data size
fread(data, sizeof(data[0]), length / (sizeof(data[0])), wavFile);       // Reading WAV data into an array
fclose(wavFile);                                                         // Closing the WAV file as data required is parsed into variables
```

Figure 3.3 The Parsing of Data from the WAV file

Once the header file is read, the formatting chunk data can be accessed to obtain the value of the length of the sampled data. This length is used to create an appropriately sized data array and also used to create the limits of the normalisation later in the code. Fseek() is used in this code example to point to the end of the header and the start of the data chunk. The size of the data array is outputted for the user also. This can be useful to a user when changing sampling rates or removing data. The final operation before the WAV file is closed is the fread(). This fread() populates the generated data array with the hexadecimal bytes from the WAV file. A benefit to using the array is that the element is stored. This allows for the little endian formatting to take place during the normalisation of the values into real data.

Section 2 – The Normalisation of the Audio Data

The final portion of the given code involves normalising the data array into a signal vector. This is shown in figure 3.4.



```
vector<double> NewSignal;                                                // Using a vector to store normalised WAV data
for (int i = 0; i < length; i = i + 2) {                                 // Data uses 2 bytes for each value
    int c = ((data[i] & 0xff) | (data[i + 1] << 8));                     // Compensating for little endian formatting LSB -> MSB
    double t;
    if (i > length / 2) {                                                // Normalising all values within the data array
        t = c / 32767.0;
    }
    else {
        t = 0.0;
    }
    NewSignal.push_back(t);                                              // Storing the normalised data
}
return 0;
```

Figure 3.4 The generation of signal values from the WAV data

The vector variable is used as it can easily hold new data by changing length dependent to the input (Friedman and Koffman, 2007, pp600). In comparison to an array, the size of which is decided upon initialisation.

All of the code in figure 3.4 happens within a for loop. This for loop compensates for the two-byte data size for each signal, the counter ‘i’ jumps two places instead of one. The two bytes also need to switch to having the most significant bit at the front of the signal before it is normalised. This decoding from little endian is done with the integer variable ‘c’.

Finally, the data is normalised into signal data by dividing the integer variables with the floating number. This creates the audio samples at the quantised rate specified within the WAV header. All the normalised signal values are then placed into the signal vector, for later usage and adaptation. The opposite of this function could also be carried out with the placement of the audio signal back into a data array and subsequent a WAV file with the header and data passed into the file.

Results

The result from this code is observed to be an audio file that has the metadata successfully parsed from the signal data of the audio waveform. The signal data has been normalised and stored in a vector data type, which can be modified using digital signal processing techniques.

Conclusion

To conclude, this code utilises a way of parsing through a WAV file. The header file is used to obtain the data chunk size and predefined lengths for the formatting and RIFF chunks. The data is successfully read and passed into an array before going into a for loop to normalise signal values. Future iterations of this code may include taking a more object orientated approach, so that code can be reused in an efficient manner.

From here, the code can be manipulated for the given assignment, changing the sample rate, and muting a section.

4. Programming Assignment

Introduction

This section is an overview to part 3 of the given assignment for the WAV file exercise. Within the assessment brief, two main modifications to the given code described in chapter 3 are changing the sampling rate of a mono WAV file and save in a new version of the file, as well as mute a section within the audio file using equation 1 below.

$$D_t\colon from\ sample\left(\frac{N}{2}+\frac{N}{100}\right)to\ sample(\frac{N}{2}+\frac{N}{30})$$

Equation 1 The Time Slot for Muting

Where Dt is the time slot and N is the maximum number of samples in the audio file.

The changes are implemented alongside changes to the given code structure so that functions are used. The approach to object orientated programming allows for reusability of code and fast adaptation (Friedman and Koffman, 2007, pp16). Further to discussion on the code, results are used to analyse the three outputted WAV files with MATLAB providing graphical representations of the WAV files.

The full code with comments describing the functions is given in Appendix 2. The focus of this report is to discuss the implementation of the changes requested and how they are implemented in C++.

Changing the Sample Rate

For this part, the changing of sample rate requires changes to both the format chunk of the WAV file and the data chunk. For the change of sample rate, up sampling is used to double the sample rate. To do this, interpolation is used to increase the sample rate (Pohlmann, 2010, pp108). The average value between audio signals is taken and placed in between the original signal.

Additionally, the header file has to reflect the change for the data to be correctly read. This includes doubling the Subchunk2Size (Data chunk size), SamplesPerSec and bytesPerSec. Figure 4.1 is a flowchart for this function. A nested for loop is used to place the average of two signals between each other. Furthermore, printing the data size of the new file allows for easy checking of data size.

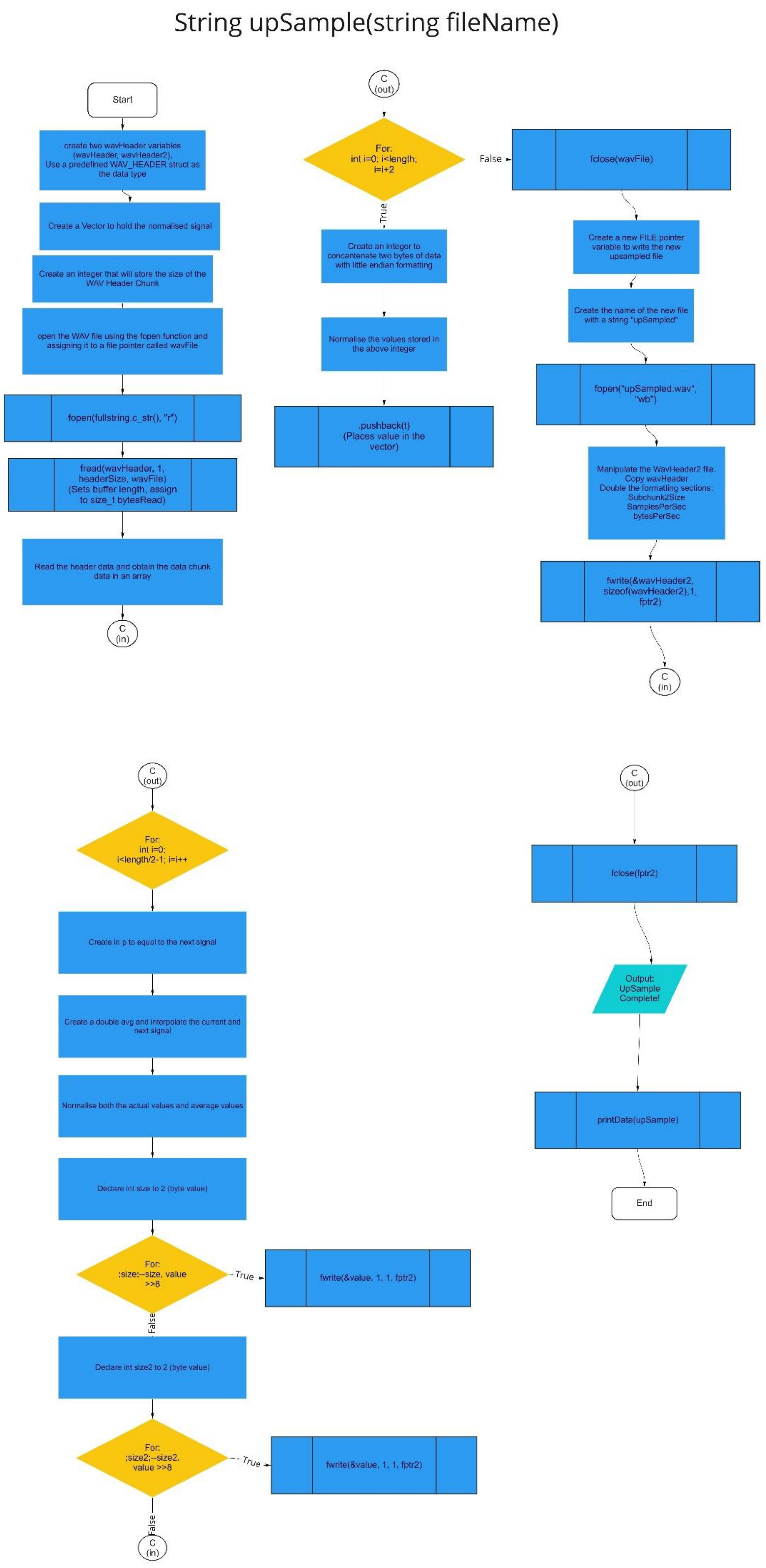
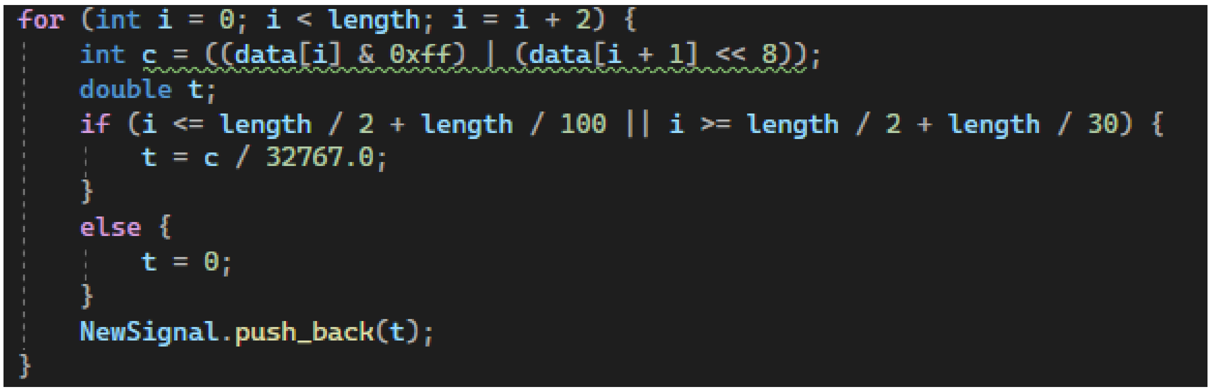

String upSample(string fileName)
Start
create two wavHeader variables (wavHeader, wavHeader2), Use a predefined WAV_HEADER struct as the data type
Create a Vector to hold the normalised signal
Create an integer that will store the size of the WAV Header Chunk
open the WAV file using the fopen function and assigning it to a file pointer called wavFile
fopen(fullstring.c_str(), "r")
fread(wavHeader, 1, headerSize, wavFile) (Sets buffer length, assign to size_t bytesRead)
Read the header data and obtain the data chunk data in an array
C (in)
C (out)
For: int i=0; i<length; i=i+2
False
True
Create an integer to concantenate two bytes of data with little endian formatting
Normalise the values stored in the above integer
.pushback(t) (Places value in the vector)
fclose(wavFile)
Create a new FILE pointer variable to write the new upsampled file
Create the name of the new file with a string "upSampled"
fopen("upSampled.wav", "wb")
Manipulate the WavHeader2 file. Copy wavHeader Double the formatting sections: Subchunk2Size SamplesPerSec bytesPerSec
fwrite(&wavHeader2, sizeof(wavHeader2),1, fptr2)
C (in)
C (out)
For: int i=0; i<length/2-1; i=i++
Create in p to equal to the next signal
Create a double avg and interpolate the current and next signal
Normalise both the actual values and average values
Declare int size to 2 (byte value)
For: ;size;--size, value >>8
True
fwrite(&value, 1, 1, fptr2)
False
Declare int size2 to 2 (byte value)
For: ;size2;--size2, value >>8
True
fwrite(&value, 1, 1, fptr2)
False
C (in)
C (out)
fclose(fptr2)
Output: UpSample Complete!
printData(upSample)
End

Figure 4.1 A Flowchart for the UpSample Function

To conclude, the up sample function created will create a second file, upSampled.wav. This fulfils the criteria for changing the sample rate of a WAV file with both the Header chunk and Data chunk changing to compensate for a doubling of values. In testing, changing the Header solely speeds the audio file to twice the speed.

Muting a Section

The muting of a section within the audio data requires placing some audio data to a value of zero. To do this, the audio WAV file from the up sample function is copied over to the mute function. When rewriting the file, the header file and data file size stays the same. However, when rewriting, an if function is used to define a muted section, shown in figure 4.2.



```
for (int i = 0; i < length; i = i + 2) {
    int c = ((data[i] & 0xff) | (data[i + 1] << 8));
    double t;
    if (i <= length / 2 + length / 100 || i >= length / 2 + length / 30) {
        t = c / 32767.0;
    }
    else {
        t = 0;
    }
    NewSignal.push_back(t);
}
```

Figure 4.2 The if Function Muting Data Values

This method, muting specific values leaves an audible blip in the file. It also allows for data to not be lost or mixed up. The muted signal is outputted to another WAV file, named mutedSample.wav. Figure 4.3 provides a flowchart for the whole function.

String muteSignal(string fileName)

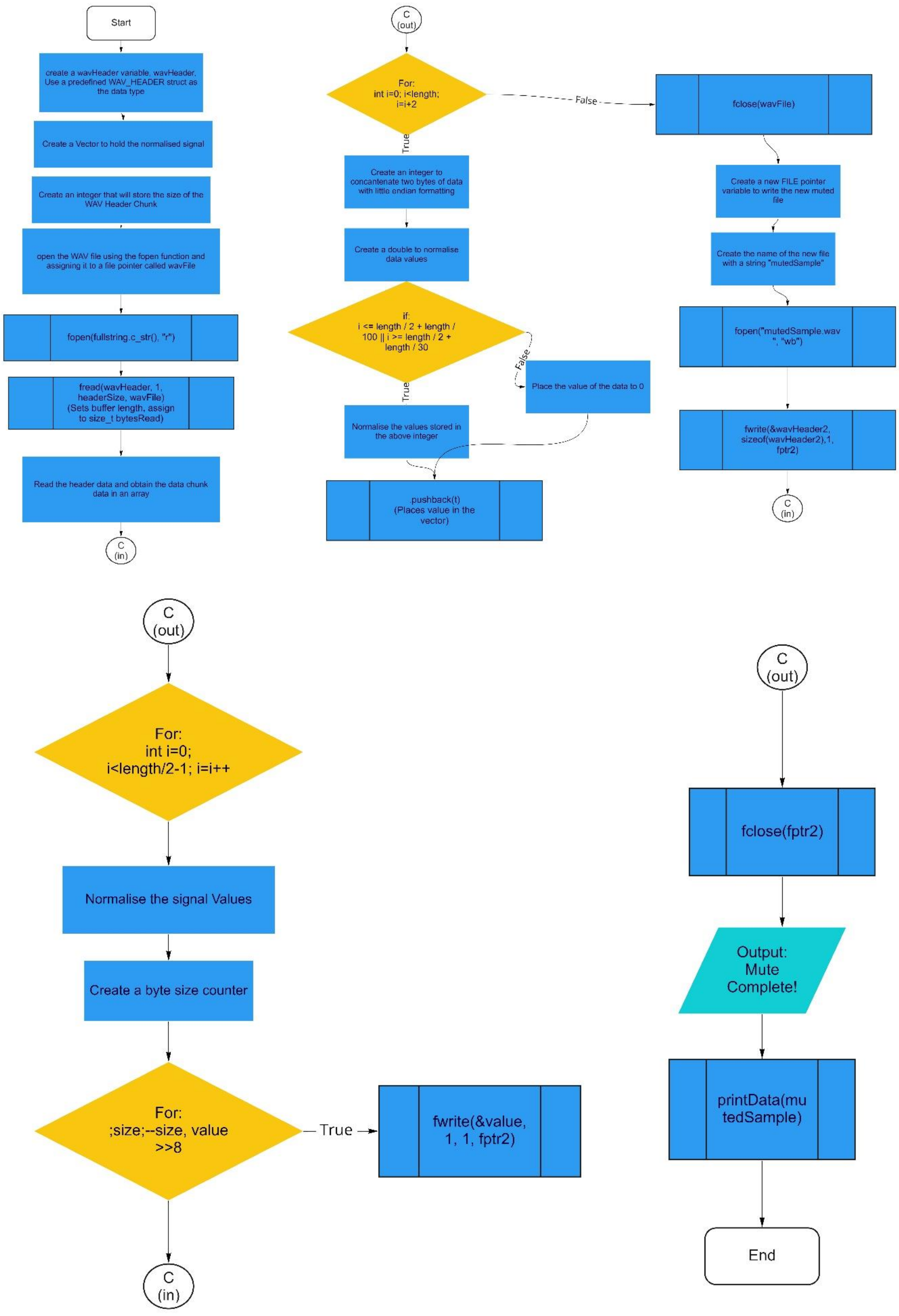

Start
create a wavHeader variable. wavHeader, Use a predefined WAV_HEADER struct as the data type
Create a Vector to hold the normalised signal
Create an integer that will store the size of the WAV Header Chunk
open the WAV file using the fopen function and assigning it to a file pointer called wavFile
fopen(fullstring.c_str(), "r")
fread(wavHeader, 1, headerSize, wavFile) (Sets buffer length, assign to size_t bytesRead)
Read the header data and obtain the data chunk data in an array
C (in)
C (out)
For: int i=0; i<length; i=i+2
False
True
Create an integer to concantenate two bytes of data with little endian formatting
Create a double to normalise data values
if: i <= length / 2 + length / 100 || i >= length / 2 + length / 30
False
True
Place the value of the data to 0
Normalise the values stored in the above integer
.pushback(t) (Places value in the vector)
fclose(wavFile)
Create a new FILE pointer variable to write the new muted file
Create the name of the new file with a string "mutedSample"
fopen("mutedSample.wav", "wb")
fwrite(&wavHeader2, sizeof(wavHeader2),1, fptr2)
C (in)
C (out)
For: int i=0; i<length/2-1; i=i++
Normalise the signal Values
Create a byte size counter
For: ;size;--size, value >>8
True
fwrite(&value, 1, 1, fptr2)
C (in)
C (out)
fclose(fptr2)
Output: Mute Complete!
printData(mutedSample)
End

Figure 4.3 A Flowchart of the Mute Signal Function

Results

This section discusses the results of the changes to the original WAV file inputted into the program. For this, MATLAB is used to plot all three files waveforms to allow for visual comparison. This is shown in Figure 4.4 below, Appendix 3 is the MATLAB code used.

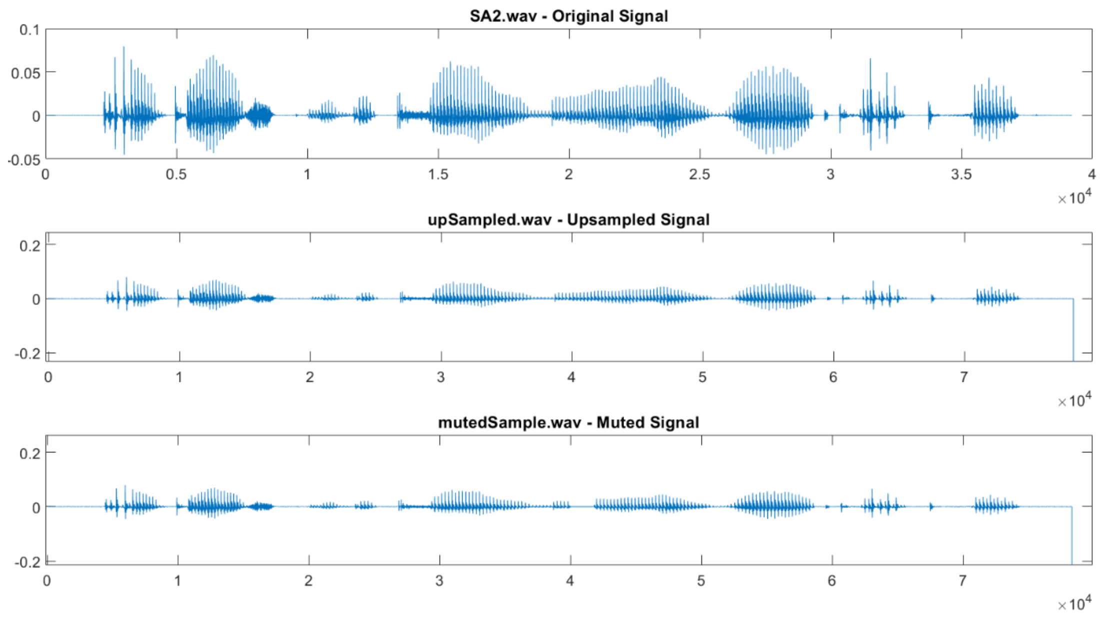


Figure 4.4 The Three Outputted WAV File Waveforms

Where the x axis represents the amplitude, and the y axis represents the sample. From figure 4.4, we see that the sample rate has doubled between 'SA2.wav' and 'upSampled.wav'. We also see a mute at the 4-sample point on the muted signal graph. This visual result confirms the audio playback heard upon playback of all three files. The visual representation from MATLAB is used to confirm the program has run correctly and satisfied the requirements for the assignment.

Upon playback, the original file and up sampled file sound very similar, with the muted sample playback giving the biggest difference audio.

Conclusion

To conclude this section, the sample rate of a WAV audio file has been manipulated by up sampling. This is done via interpolation, obtaining the average of two signals and placing the data value between the original values. A mute is implemented by commuting a section to a zero value. This zero value portion is defined in equation 1, with results of both highlighted in figure 4.4.

Future work on this program includes the usage of user input. At the current stage, no user input is used, although using file checking. Future work would implement this feature. Furthermore, a header file could be used to store the WAV Header Chunk information to declutter the program.

5. Conclusion

Discussion and Findings

The findings of the research in chapter 2 highlight the format of the WAV file as a resource interchange file format (RIFF) type to hold both the audio data and metadata to recreate the audio in programs. The format to which holds the file formatting, the data signal processing details and finally the data.

From chapter 2 research, the usage of little endian formatting on Microsoft and IBM processors is implemented in both chapter 3 and 4 programming. The understanding of the implementation allowed for successful programs for inputting & outputting WAV files withing C++.

The findings of the programming assignment (Chapter 4) is the successful implementation of changing the sample rate by both updating the header chunk of a WAV file and interpolation of data within a WAV file.

Findings

The WAV audio format provides a way of supporting the audio data and formatting details in a lossless file format. This file can be manipulated using signal processing techniques. Furthermore, the WAV file format is usable across multiple devices and can be accessed in C++ with understanding of the data type.

Bibliography

Bird, J. (2017). Engineering Mathematics. Milton: Taylor and Francis.

Dempsey, J. (2020). How Do MP3 and WAV Files Differ? [online] Dawsons Music. Available at: https://www.dawsons.co.uk/blog/how-do-mp3-and-wav-files-differ.

Friedman, F.L. and Koffman, E.B. (2007). Problem solving, abstraction, and Design Using C++. 5th ed. Boston: Pearson.

Geeks for Geeks (2021). Little and Big Endian Mystery. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/little-and-big-endian-mystery/.

Microsoft and IBM (1991). Multimedia Programming Interface and Data Specifications 1. United States: Microsoft Corporation.

Pohlmann, K.C. (2010). Principles of Digital Audio. New York: Mcgraw Hill Professional.

Appendix 1 – Week 11 Lecture Code

```
// Harriet Drury 26/01/2022
// Provided Code to Read a WAV file
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <cstdint>
#include "stdio.h"
using namespace std;

struct WAV_HEADER {
	uint8_t		RIFF[4];		// RIFF Header Magic header
	uint32_t	ChunkSize;		// RIFF Chunk Size
	uint8_t		WAVE[4];		// WAVE Header
	uint8_t		fmt[4];			// FMT header
	uint32_t	Subchunk1Size;		// Size of fmt chunk
	uint16_t	AudioFormat;	// Audio format 1=PCM,6=mulaw,7=alaw
	uint16_t	NumOfChan;		// Number of Channels 1=mono, 2
=stereo
	uint32_t	SamplesPerSec;		// Sampling Frequency in Hz
	uint32_t	bytesPerSec;	// bytes per second
	uint16_t	blockAlign;		// 2=16-bit mono, 4=16-bit stereo
	uint16_t	bitsPerSample;		// Number of bits per sample
	uint8_t		Subchunk2ID[4];		// "data" string
	uint32_t	Subchunk2Size;		// Sampled data length
} wav_hdr;

int main() {

	WAV_HEADER wavHeader;
					// Creating a wav header struct
	int headerSize = sizeof(wav_hdr), filelength = 0;
			// Parsing the amount of data required for the header into an
int
	FILE* wavFile = fopen("melody_mono.wav","r");
			// Opening the WAV file
	fread(&wavHeader,1,headerSize, wavFile);
				// Reading the data in chunks


	int length = wavHeader.Subchunk2Size;
				// Provides length of sampled data, used to read the
data into a vector for manipulation
	int8_t* data = new int8_t[length];
					// Creating the data lenth
	fseek(wavFile, headerSize, SEEK_SET);
				// Setting file pointer to start at data chunk

	cout << sizeof(data[0]) << endl;
				// Reading the data size
	fread(data, sizeof(data[0]), length / (sizeof(data[0])), wavFile);
		// Reading WAV data into an array
	fclose(wavFile);
					// Closing the WAV file as data required
is parsed into variables

	vector<double> NewSignal;
					// Using a vector to store normalised WAV data
	for (int i = 0; i < length; i = i + 2) {
				// Data uses 2 bytes for each value
		int c = ((data[i] & 0xff) | (data[i + 1] << 8));
			// Compensating for little endian formatting LSB -> MSB
		double t;
		if (i > length / 2) {
						// Normalising all values within the data
array
			t = c / 32767.0;
		}
		else {
			t = 0.0;
		}
		NewSignal.push_back(t);
						// Storing the normalised data
	}
	return 0;
}
```

Appendix 2 – The Programming Assignment Code

```
//Harriet Drury - 26/01/2022
//A programme to take an inputed Mono audio file and output two files, one with a
doubled sample rate
//and another with a muted section

#pragma warning(disable:4996)          // For usage in visual studio, stops an
fopen error. Remove if not using visual studio

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <cstdint>
#include "stdio.h"
using namespace std;

struct WAV_HEADER {
	uint8_t		RIFF[4];			// RIFF Header Magic header
	uint32_t	ChunkSize;			// RIFF Chunk Size
	uint8_t		WAVE[4];			// WAVE Header
	uint8_t		fmt[4];				// FMT header
	uint32_t	Subchunk1Size;		// Size of fmt chunk
	uint16_t	AudioFormat;	// Audio format 1=PCM,6=mulaw,7=alaw
	uint16_t	NumOfChan;			// Number of Channels 1=mono, 2
=stereo
	uint32_t	SamplesPerSec;		// Sampling Frequency in Hz
	uint32_t	bytesPerSec;	// bytes per second
	uint16_t	blockAlign;			// 2=16-bit mono, 4=16-bit stereo
	uint16_t	bitsPerSample;		// Number of bits per sample
	uint8_t		Subchunk2ID[4];		// "data" string
	uint32_t	Subchunk2Size;		// Sampled data length
} wav_hdr;

bool checkWav(string fileName){
				// A function to check the file is correct
	string ending = ".wav";
	string fullString = fileName + ending;
		// Used to place into fopen function
	bool found;

	FILE* wavFile = fopen(fullString.c_str(), "r");
		// Casting to .c_str() for functionality of fopen
	if (!wavFile) {
					// Checks that the WAV file can be opened
		cout << "\nThe file could not be opened" << endl;
		exit(1);
		fclose(wavFile);
		found = false;
					// Sets the check boolean
	}
	else {
		cout << "\t\tFile Found - Success!" << endl;
		cout << "\t\t----------------------" << endl;
		found = true;
					// Used in main to carry out upsampling and
muting
		fclose(wavFile);
		cout << "Performing Upsampling and Muting....." << endl;
	};
	return found;
}
```

```
void printData(string fileName) {
                    // Printing the required/changed values within the header
file
      WAV_HEADER wavHeader1;
                          // Declaring variables
      int headerSize = sizeof(WAV_HEADER);
      string ending = ".wav";
      string fullString = fileName + ending;
      FILE* wavFile = fopen(fullString.c_str(), "r");

      //Read the Header

      size_t bytesRead = fread(&wavHeader1, 1, headerSize, wavFile);         //
reading the head file data and storing in a buffer
      fclose(wavFile);
      cout <<"\t\t" << fileName << " File Data" << "\n";
      // Outputting required parts from Header
      cout << "RIFF Header                   :" << wavHeader1.RIFF[0] <<
wavHeader1.RIFF[1] << wavHeader1.RIFF[2] << wavHeader1.RIFF[3] << '\n';
      cout << "WAVE Header                   :" << wavHeader1.WAVE[0] <<
wavHeader1.WAVE[1] << wavHeader1.WAVE[2] << wavHeader1.WAVE[3] << '\n';
      cout << "FMT                    :" << wavHeader1.fmt[0] <<
wavHeader1.fmt[1] << wavHeader1.fmt[2] << wavHeader1.fmt[3] << '\n';
      cout << "Channels               :" << wavHeader1.NumOfChan << '\n';
      cout << "Samples                       :" << wavHeader1.SamplesPerSec <<
'\n';
      cout << "Data Size              :" << wavHeader1.Subchunk2Size << endl;
      cout << endl;
}

string upSample(string fileName){
                    // The change of sampling rate. Doubles the sample rate with
interpolation
      WAV_HEADER wavHeader, wavHeader2;
                    // declares Header files for original file and new file
      vector<double> NewSignal;
                    // A vector to store the Data chunk
      int headerSize = sizeof(WAV_HEADER);
             // obtaining the buffer size
      string ending = ".wav";
      string fullString = fileName + ending;
      FILE* wavFile = fopen(fullString.c_str(), "r");
             // Obtains required information from the first file

      //Read the Header
      size_t bytesRead = fread(&wavHeader, 1, headerSize, wavFile);
      int length = wavHeader.Subchunk2Size;
             // Using data size from file
      int8_t* data = new int8_t[length];                  //8 bits signed
pointer
      fseek(wavFile, headerSize, SEEK_SET);
      fread(data, sizeof(data[0]), length / (sizeof(data[0])), wavFile);
      for (int i = 0; i < length; i = i + 2) {
             int c = ((data[i] & 0xff) | (data[i + 1] << 8));
      // Little Endian formatting
             double t;
             t = c / 32767.0;
                          // Normalisation
             NewSignal.push_back(t);
                          // Writing the data to the vector for manipulation
      }
      fclose(wavFile);

      // output data to a new wav file
      FILE* fptr2; // File pointer for WAV output
      string upSample = "upSampled";
```

```
	fptr2 = fopen("upSampled.wav", "wb"); // Open wav file for writing
	wavHeader2 = wavHeader;
				// Manipulating the header file data with the change in
sample rate
	wavHeader2.Subchunk2Size = wavHeader2.Subchunk2Size * 2;		//
upsampling by 2
	wavHeader2.SamplesPerSec = wavHeader2.SamplesPerSec * 2;
	wavHeader2.bytesPerSec = wavHeader2.bytesPerSec * 2;

	fwrite(&wavHeader2, sizeof(wavHeader2), 1, fptr2);
	// Writing the new header file to the generated WAV file

	for (int i = 0; i < length / 2 - 1; i++) {
		int p = i + 1;
		double avg = (NewSignal[p] + NewSignal[i]) / 2;
		// Taking the average between points to create new data
		int value = (int)(32767 * NewSignal[i]);
		// Normalise to integers
		int avgValue = (int)(32767 * avg);
			// Normalise avg
		int size = 2;
				// little endian formatting
		for (; size; --size, value >>= 8) {
			// Writes the original data
			fwrite(&value, 1, 1, fptr2);
		}
		int size2 = 2;
		for (; size2; --size2, avgValue >>= 8) {
		// Writes the interpolated data
			fwrite(&avgValue, 1, 1, fptr2);
		}
	}
	fclose(fptr2);
	cout << "UpSample Complete!" << endl;
	printData(upSample);
				// Used to check the upsample was correct
	return upSample;
}

string muteSignal(string fileName) {
	WAV_HEADER wavHeader;
				// Creating the WAV Header
	vector<double> NewSignal;
			// Creating the Storage for new Data including the Mute
	int headerSize = sizeof(WAV_HEADER);
		// Opening the Header file data
	string ending = ".wav";
	string fullString = fileName + ending;
	FILE* wavFile = fopen(fullString.c_str(), "r");
		// Opening the file

	size_t bytesRead = fread(&wavHeader, 1, headerSize, wavFile);
			// Reading the Header Chunk
	int length = wavHeader.Subchunk2Size;
				// Finding the length of the Data Chunk
	int8_t* data = new int8_t[length];
					// 8 bits signed pointer
	fseek(wavFile, headerSize, SEEK_SET);
				// Pointing to the Data portion
	fread(data, sizeof(data[0]), length / (sizeof(data[0])), wavFile);
			// Reading the Data to an array
	for (int i = 0; i < length; i = i + 2) {
		int c = ((data[i] & 0xff) | (data[i + 1] << 8));
		double t;
		if (i <= length / 2 + length / 100 || i >= length / 2 + length / 30)
{		// Performing the mute with the required section
```

```
                        t = c / 32767.0;
                                                                    // Normalisation
                }
                else {
                        t = 0;
                                                                            // Muting by commuting to 0
                }
                NewSignal.push_back(t);
                                                                    // Placing normalised data into the
vector
        }
        fclose(wavFile);

        string mutedSample = "mutedSample";
        FILE* fptr; // File pointer for WAV output
        fptr = fopen("mutedSample.wav", "wb"); // Open wav file for writing


        fwrite(&wavHeader, sizeof(wavHeader), 1, fptr);

        for (int i = 0; i < length /2 - 1; i++) {
                int value = (int)(32767 * NewSignal[i]);                // Normalise to
integers
                int size = 2;
        // little endian formatting
                for (; size; --size, value >>= 8) {
                        fwrite(&value, 1, 1, fptr);
                }
        }
        fclose(fptr);
        cout << "Mute Complete!" << endl;
        printData(mutedSample);
                                                                    // To check the writing to file was
correct
        return mutedSample;
}


int main()
{
        string fileName = "SA2";
                                                            // Change when needed, used to input the
WAV file in the same directory as the application
        bool found = checkWav(fileName);
                                                    // Checking validity
        if (found == true) {
                                                                    // Running the upsampling and
muting if valid file
        printData(fileName);
        muteSignal(upSample(fileName));
        }
        else { cout << "Check the File Name is Correct and in the correct folder";
}       // Providing an error message
        return 0;
}
```

Appendix 3 – MATLAB Results Code

```matlab
%Harriet Drury - 26/01/2022
%analysing the outputted Wav files in Comparison to the inputted file

x = audioread('SA2.wav');
y = audioread('upSampled.wav');
z = audioread('mutedSample.wav');

figure(1);
subplot(3,1,1)
plot(x)
title ('SA2.wav - Original Signal');    %Giving a title
ax = gca;                          %Setting the fontsize to a legible size
ax.FontSize = 16;
subplot(3,1,2)
plot(y)
title ('upSampled.wav - Upsampled Signal');    %Giving a title
ax = gca;                          %Setting the fontsize to a legible size
ax.FontSize = 16;
subplot(3,1,3)
plot(z)
title ('mutedSample.wav - Muted Signal');    %Giving a title
ax = gca;                          %Setting the fontsize to a legible size
ax.FontSize = 16;
```