Graphical Audio Waveform Display

A Project by

Hannah Katrina B. Cariño
Ma. Kassandra D. Salvacion

Submitted to

Luisito L. Agustin
Instructor, CE 101

In Partial Fulfillment of the Requirements for the Course
CE 101: Techniques in Signal Processing

Department of Electronics, Computer and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

October 2010
Abstract

This project implements a waveform display for an audio file using wxDev-C++.

An audio file can be loaded to the program, where it validates, reads, and obtains relevant information from the chunks in the WAV file according to the RIFF structure. To keep things simple, the program deals only with uncompressed PCM (compression code 0) WAV files, but deals with both audio and stereo files. Data from the data chunk are loaded into two dynamic arrays or vectors, which plot the points with the help of koolplot, an open source graphing library. A waveform across the time domain is produced and displayed separately for each audio file loaded.

The program has not fully developed the graphic user interface of the input stream of the loaded file, however, the group succeeded in creating a program that allows multiple window display for the audio files loaded.
Acknowledgments

This project would not be made possible without the following people;

To our parents and family, who have supported us all throughout the sleepless nights; they have provided us food, shelter, company, entertainment and most of all patience in our moments of temporary insanity;

To our friends who have been there all the way, praying for us and giving us encouragement that we will be able to get through the hardships amidst the physical and mental weariness;

To our CH-ACS super friends who shared the same pain, laughter, tears and joys not only in creating this project but all throughout each and every endeavor in this very magical journey of our (hopefully) last year in school;

To Margo, our dear friend, who will never be forgotten and who is still here with us every single step of the way;

To football matches to keep us awake at the wee hours of the morning, to the library sessions that have witnessed our successes and failures, our progress and regresses all throughout;

To the Salvacion family, for our endless supply of ensaymada, Chips Ahoy, water, food, electricity consumption and of course, their never-ending hospitality to the sleepless nights we have spent on this program.

To Sir Lui, who had been our guidance and mentor and who gave all of his support;

And to God to whom all things our glorified and whose will is fulfilled through everything in this world;

Our deepest and utmost gratitude, for because of all of you, we have come this far.
# Table of Contents

1. Introduction                                                                  3
   1.1. Objective                                                               3
   1.2. Scope and Limitations                                                  3

2. Review of Related Literature                                                  10
   2.1. Sound and Digital Audio                                               10
   2.2. The WAV File Format                                                   12
       2.2.1 WAV File Header                                                  12
       2.2.2 WAV File Chunks                                                  12
           2.2.2.1 Format Chunk - "fmt"                                      12
           2.2.2.1.1 Chunk ID and Data Size                                 13
           2.2.2.1.2 Compression Code                                      13
           2.2.2.1.3 Number of Channels                                    13
           2.2.2.1.4 Sample Rate                                           13
           2.2.2.1.5 Average Bytes Per Second                               13
           2.2.2.1.6 Block Align                                           13
           2.2.2.1.7 Significant Bits Per Sample                            14
           2.2.2.1.8 Extra Format Bytes                                     14
           2.2.2.2 Data Chunk - "data"                                      14
           2.2.2.3 Fact Chunk - "fact"                                      14
           2.2.2.4 Wave List Chunk - "wavl"                                  15
   2.3 Pulse Code Modulation                                                    15

3. Audio Waveform Display                                                       15
   3.1. Program Overview                                                       16
       3.1.1. Data Extraction                                                 17
           3.1.1.1. Standard Headers Used                                   17
           3.1.1.2. Classes Used                                           18
3.1.1.2.1. WAV_IN class .......................................................... 19
3.1.1.2.2. WAV_OUT class ...................................................... 21
3.1.2. Waveform Plotting ....................................................... 21
3.1.2.1. plotdata ................................................................. 21
3.1.2.2. plot() ...................................................................... 21
3.1.2.3. addMark() ............................................................... 22
3.1.2.4. axesBotLeft(); axesTopRight ................................. 23
Appendix 1. Software Source Codes ........................................ 25
A.1. driver.cpp ...................................................................... 25
A.2. wav_out.cpp .................................................................. 26
A.3. wav_in.cpp .................................................................... 30
A.4. common.h ..................................................................... 34
A.5. f_err.h ......................................................................... 38
A.6. f_ptch.h ....................................................................... 38
A.7. wav_def.h ..................................................................... 38
A.8. wav_in.h ....................................................................... 39
A.9. wav_out.h ..................................................................... 40
References ........................................................................... 41
Introduction

1.1. Objective

The focus of the project is mainly about recreating a functional program that is able to produce an audio waveform display from an input audio file. As this had been previously done by Begornia et al., the project serves as a continuation of the previous project with the general aim of improving its features particularly its graphic user interface (GUI).

The general objective of the project then is to be able to recreate the interface that enables multiple windows to be opened simultaneously. Given an audio file, the program must display its respective audio waveform plot upon loading. When a different audio file is loaded, the program must retain the previous audio waveform plot of the first loaded audio file while being able to display in a separate window the audio waveform plot of the other loaded audio file. With the given program, this feature is disabled as it only allows one working window for every loaded audio file input.

Once the interface is recreated and the general objective is obtained, the group aims to add more features to the newly revised program such as try to successfully load other audio file formats such as .mp3, if time permits so. The previous project had concentrated on loading the common audio file format, the .wav format. The additional features however, are only secondary to the primary objective of the project and might be scrapped depending on time constraints.

1.2. Scope and Limitations

As the group had used the same program framework as the creators of the program, no major changes had been made in the features of the program and therefore the same scope and limitations apply.
From the previous work, the program is designed to function only and specifically to .wav file formats, as it is one of the most common audio file format used, and to simplify things first with uncompressed data. The .wav audio files that can be loaded with the program functioning properly are limited to either 8-bit or 16-bit sample form in either mono (single channel audio file) or stereo (two channel audio file).

Once the audio file input loads, the program displays the waveform instantaneously in a separate child window. There are only a number of features that can be used to manipulate the waveform display. The list of features include a clear quality of the plot that is currently being viewed, zooming both in and out, panning from left to right and changing the color of the audio waveform being displayed. The details of the audio file can also be viewed and displayed such as the file format, time divisions, sampling rates, byte rate, audio type etc. [1]

The graphical user interface is also maintained from the previous work with minor modifications to fit the objectives of the project. Aside from the minor modification, the user interface was mostly retained with limitations on the movement on the child window of the audio waveform displayed. The audio waveform display window has no equipped scrollbar or track bar, and the user cannot configure the size of the window manually. The display window changes its size accordingly with the loaded audio input. For mono audio input file loading, the size of the window is halved. The original size of the window is being retained for the case of stereo audio input file loading.

The interface must also controls to help users navigate throughout the program such as the mouse buttons, arrow keys in the keyboard and three customized parent menus containing common commands used in programs. The first drop-down parent menu is used for loading the input file, and also offers an alternative option for exit. In the case of another loading, the same command can be used and the program performs accordingly. The second drop-down menu is for the aesthetic and overall manipulation of the waveform display as it contains commands where the color of the waveform display
can be changed according to the user’s desired option among the six default colors. The last drop-down menu provides information about the program including an overview, controls, features, about the authors and acknowledgment that will be displayed in separate windows when opened.

The main limitation of the program is the size of the input file that can be loaded into the program without crashing. Due to the complexity of programming required for larger file sizes, the original programmers decided to limit the file size that can be inputted into the program that will display an audio waveform successfully.

The other limitation is the audio file format that can be loaded into the program. The creators of the program mainly focused on loading audio files in .wav format with the possibilities of loading other file formats left for future programmers to tackle on.
2. Review of Related Literature

2.1. Sound and Digital Audio

The concept of sound is defined to be fluctuations in air pressure caused by vibration of objects. When an object moves, it displaces the air molecules next to it and this displacement results in a high-pressure front that travels towards a human’s ear, enabling the “sound” to be heard. Human ears, however can only hear sounds at a specific frequencies ranging from 20 to 20,000 Hz.

Sounds are analog signals and are real-time and therefore cannot be captured in a physical sense. Over time, however, technology has developed several ways to capture sound and convert them into electrical signals to allow operations to be performed on it such as amplification, recording and mixing. [2] The most common method for sound preservation is through recording. Sound recording is the inscription of sound waves through mechanical or electrical means. Sound recording can either be analog or digital wherein the former stores sound through a physical texture such as a phonograph. Digital recording differs from analog recording as digital recording includes the conversion of audio analog signals into discrete numbers, which serves as a digital approximation. [3]

The process of digital recording involves the discrete sampling of a sound wave’s amplitude and store the information to be later reproduced at the same rate to recreate the sound wave. The changes in air pressure are converted into voltages through a transducer and an analog-to-digital converter device receives these discrete voltages, assigning values to each amplitude. The process of converting voltages to discrete numbers (in the computer, a string of binary numbers 1 or 0) is called quantization. A complement device called a digital-to-analog converter is used to play the sound back at the same rate through reading the numbers from memory. In a reverse process, these values are converted back to an equivalent voltage that is communicated to an amplifier to increase the amplitude. [4]
In an electronic device such as a computer, digital audio is stored in different formats. Audio file formats are container formats or audio file formats with a defined storage. These file formats can either be compressed or uncompressed file formats, which varies depending on the desired file size. The most common audio file formats include .mp3, .mid, .aif and the .wav format. [5]

2.2 The WAV file format

WAV, for Waveform Audio Format, file is an audio file format, developed by Microsoft. It has become one of the most widely supported digital audio FPC file formats due to the popularity of and wide range of programs written for the Windows platform. Through the years, the WAV file has expanded to become a viable interchange medium for other platforms, including Macintosh that allowed interoperability of audio files in different platforms for processing.

The WAV file format is a lossless format. This means that the compression algorithm used allows reconstruction of exact original data from the compressed data. Consequently, these files are larger compared to files saved in lossy formats, that are smaller in size at the expense of audio quality. Thus, WAV file formats are most often used for archiving purposes where quality of data is retained. Additionally, these files are commonly used as a master to create other types of lossy compression audio files. [6]

The said format is based on the Resource Interchange File Format (RIFF), which stores files in indexed "chunks" and "sub-chunks". The structure groups the file contents, including sample format, digital audio samples, etc., into separate chunks that contain its own header data and bytes. The chunk header defines the type and the size of the chunk data bytes. These allow programs to parse through the file and locate only the necessary chunks for the said program. Further discussion of these chunks and sub-chunks for the audio manipulation program are described below. [6]
It is important to note that RIFF file chunks should necessarily be word aligned. Their total size must be in multiples of 2 bytes; otherwise, extra padding bytes with a value of zero should be added. Additionally, these padding bytes should be noted in order to calculate the offset of the following chunk. [7]

2.2.1 WAV File Header

As based on the standard RIFF structure, the first 8 bytes of the file is the RIFF chunk header. The first four bytes contain the chunk id of "RIFF" followed by another 4 bytes containing the file size (minus the file header). The succeeding four data bytes after the file size contain the type of resource in the RIFF chunks, which in this case, being WAV files, contain the id “WAVE". The WAV file chunks that characterize the audio waveform follow this chunk.

2.2.2 WAV File Chunks

WAV files are generally characterized by two chunks, the Format Chunk and Data Chunk. These two chunks are necessary to describe the format of the digital audio samples and the sample themselves. It is important to note that many programs place the Format Chunk before the Data Chunk as it is more sensible when streaming digital audio from a slow linear source. [14] Otherwise, all of the data should have to be downloaded first to properly play the said file.

The RIFF Chunk Header glimpses into the how the said format arranges the data in individual chunks. The first four bytes of a chunk are allotted to the chunk id, while the following 4 bytes contain the size of the chunk. The remaining bytes are dedicated to the data stored in the chunk.

2.2.2.1 Format Chunk - "fmt"

The format chunk specifies the information needed to determine audio
manipulation of the file. It contains data on the type of compression used, the sample rate, bits per sample, the number of channels and other set values.

2.2.2.1.1 Chunk ID and Data Size

The chunk ID is "fmt" (0x666D7420). [8] The size indicated is the size of the standard wave format data added with the size of the extra format bytes.

2.2.2.1.2 Compression Code

This defines the compression method used on the wave data included. For simplicity's sake, the groups have tackled only files with compression code 1, or uncompressed PCM data.

2.2.2.1.3 Number of Channels

This chunk checks the number of specified separate audio signals that are in the wave data chunk. A mono signal would be indicated by a value of 1. A stereo signal is denoted by a 2.

2.2.2.1.4 Sample Rate

This is the number of sample slices obtained per second.

2.2.2.1.5 Average Bytes Per Second

The value obtained in this chunk pertains to the number of bytes per second needed to be streamed to a digital-to-analog converter to play the wave file. It can be indirectly obtained through the formula,

\[ \text{AvgBytesPerSec} = \text{SampleRate} \times \text{BlockAlign} \]
2.2.2.1.6 Block Align

The value indicates the number of bytes per sample size, and can be calculated by the formula,

\[ \text{BlockAlign} = \frac{\text{SignificantBitsPerSample}}{8} \times \text{NumChannels} \]

2.2.2.1.7 Significant Bits Per Sample

This value specifies how many bits are used to define each sample. It is usually in multiples of 8. Otherwise, the number of bits used per sample should be rounded up to the nearest byte size using padded bits with zero values.

2.2.2.1.8 Extra Format Bytes

The value notes the number of additional format bytes that follow. This block only appears if the data is compressed; the value would depend on what compression information is needed to decode the wave data. [7] The value is usually a multiple of 2; otherwise, padding with zeros is applied to align it.

2.2.2.2 Data Chunk - "data"

This chunk contains the digital audio sample data. A single data chunk is usually applicable to a single WAV file. However, it may still contain more than one, and these are listed within a WaveListChunk format. Multichannel (e.g. stereo) samples are stored by cycling through the audio samples for each channel before advancing to the next sample time, to be able to play the file even if it is not completely read. [9]
It is important to note that sample data are represented with unsigned 8 bit values, while all other sample bit-sizes are specified as signed values. Additionally, like previous chunks, these should be word-aligned. [10]

2.2.2.3 Fact Chunk - "fact"

This chunk contains information that is needed to decompress WAV files that are not uncompressed PCM forms (compression code 1).

2.2.2.4 Wave List Chunk - "wavl"

Other programs commonly disregard this chunk as it simply defines the alternating "slnt" and "data" chunks.

2.3 Pulse Code Modulation

Pulse code modulation is a process in which the standard values of a quantized wave are described by a series of coded pulses and these codes can be binary, ternary or $n$-ary. Pulse-modulation systems allows for the conversion of analog wave shapes to digital wave shapes. Given an analog wave, the entire range of the amplitude (or frequency or phase in some cases) can be divided and assigned to a series of standard values to represent samples of the wave. Each sampled point, denoted as a pulse, acquires the standard value nearest to the actual value when modulated. The accuracy of this method depends on the number of standard levels used. The greater the number of standard levels, the more accurate approximation is obtained as gaps between the standard values and the actual values are lessened [11].

For digital audio, PCM format is among the most common forms for WAV files, specifically called Linear Pulse Code Modulation (LPCM). LPCM encoding is used as the standard audio file format for audio compact disks and even in digital telephone systems. [12]
2.4 Audio Waveform Display

Representing audio visually through waveforms are helpful as they make audio file inputs distinguishable from other inputs. Waveform display also allows for non-linear editing and processing such as deleting parts of the file or manipulating them. [13]

Building an audio waveform starts from converting the raw data from the stream into audio samples that are organized by channels. The data chunk is extracted from the audio file and decoding the data chunk is to be done through the audio file details such as byte rate and the sampling rate. [1] After the data chunk is processed and divided into smaller bytes of data, it is interpreted as a series of amplitudes through an algorithm. Through an algorithm, these amplitudes can be plotted to create the audio waveform display. [15]
3. Audio Waveform Display

3.1. Program Overview

The process of the conversion of the said program from a WAV file to an audio waveform is completed in two steps: data extraction, and waveform graphing.

3.1.1. Data Extraction

The process of data extraction for the WAV file is not limited to simply reading and loading the bytes which compromise the audio files. In fact, to properly load a WAV file with the intention of audio manipulation, one must extract the bytes of the WAV file with full knowledge and interpretation of the RIFF structure the WAV file follows. These refer to the knowledge of the placement of chunks and chunk identifiers in the WAV file. These bytes are consequently stored according to their specifications. The data obtained is then put into samples, where it can finally be used for manipulation.

The said procedure is implemented by the program through first extracting a directory by buffering it, then the strings comprising the directory are placed in a character buffer which is passed on to the Data extraction algorithm for validation of the said file format and for the interpretation of each byte chunk. The values are stored in vectors, or dynamic arrays, where information such as the bits per sample and number of channels is used in the processing of data. Individual data bytes are grouped according to the allowable bits per sample information and stored in another vector, which is used for the graphing algorithm that is done in this program.

3.1.1.1. Standard Headers Used

The following headers were used in facilitating the program.

```c
#include <cstdio>,
```
This "standard input/output header" contains macro definitions, constants, and declarations of functions and types used for various standard input and output operations [18].

```c
#include <iostream.h>
```

This is a header file used for input/output in the C++ programming language. It is part of the C++ standard library. The name stands for **Input/Output Stream**. Similar to the `cstdio` header inherited, this provides basic input and output services for the program through standard streams. This header also allows for the buffering of data in the program, specifically in the data extraction.

```c
#include <cstdlib>
```

The general purpose standard library includes functions involving memory allocation, process control, conversions and others. This is important to allow the program to allot memory locations for the data read in each chunk of the WAV file.

```c
#include <cmath>
```

This header file, which is also included in the standard library of C++, is designed for basic mathematical operations. Most of the functions involve the use of floating point numbers.

### 3.1.1.2. Classes Used

#### 3.1.1.2.1. WAV_IN class

The class WAV_IN is the class created mainly for the extraction of the WAV file. This class is created as a friend to the WAV_OUT class. This class contains the following public members:

```c
double read_current_input();
```
This function allows for the reading of the data chunk of the loaded WAV file, and returns the content as a variable of *double* type. The data obtained from this algorithm is loaded into the y-values for the graphing algorithm.

```c
int more_data_available();
```

This function returns only a 0 and 1, whether it has detected the end of the file or not. If called, a 1 is sent back if there are still remaining data left to be read, while a 0 is sent if the end of the file has already been reached.

```c
long int get_num_samples();
```

A variable of *long int* type is returned for this function. The variable corresponds to the number of samples in the WAV file.

```c
int get_num_channels();
```

The number obtained from this function corresponds to the number of channels used in the audio track. Typically, a WAV file is either mono or stereo. Mono implies that there is only a single signal path for the track. Meanwhile, a track that is stereo would return a two, since a stereo track usually makes use of two (or more) independent audio channels that are configured in a way to improve sound quality e.g. by creating the impression of sound heard from various directions.

```c
int get_bits_per_sample();
```

This function reports the number of bits per sample of the file.

```c
double get_sample_rate_hz();
```
This function reports the sample rate or sample frequency, in units of Hertz (Hz).

A constructor corresponding to this class was also created with the WAV filename as the only argument. This constructor handles the loading of the input file, the validation of each individual chunks and their headers, extraction of important chunk information, and finally, extraction of the points used from the data chunk to plot the corresponding audio waveform.

3.1.1.2.2. WAV_OUT class

The class WAV_OUT is the class created to serve as a buffer for the inputted WAV file where the graphing algorithm obtains the relevant data needed. This class is created as a friend to the WAV_IN class.

```cpp
int WAV_OUT::write_current_output(double ooo);
```

This serves as the main function of this class. A buffer is allocated to the loaded input file and expanded depending on the size of the WAV file. The values are kept here until the graphing algorithm is able to plot all of the points [19, 20].

3.1.2. Waveform Plotting

In order to display the graphical audio waveform, a software library called koolplot was used. koolplot is a freeware open-source that is able to draw 2-dimensional graphs from either C or C++ programs. The source code for the koolplot library is available in the internet without any restrictions on license. It is also available for the MingW (GCC port) compiler and is functioning based on the WinBGIm library. [21]

A simple snippet on how the koolplot library is used is shown below;

```cpp
#include "koolplot.h"

int main()
{
```
Plotdata x(-3.0, 3.0), y = sin(x) - 0.5*x;
plot(x, y);
return 0;
}

3.1.2.1. plotdata

The main class used in the koolplot library is plotdata. This class is mainly used for plotting graphs. A declared plotdata variable can hold any number of data points having double values. An exception however, is a special value called NOPLOT which is never plotted into graphs.

Initialization of a plotdata variable can be done in a single argument by giving the variable a specific range i.e. a certain numerical value that would automatically be created in the plot. The initialization can also be done by declaring an existing array [22].

In the program, the initialization of the variables x and y were done simply in the manner;

plotdata x,y;

This declaration of plotdata variables allow for a more flexible range of values as the number of samples to be plotted depends on the .wav file that is being loaded.

3.1.2.2. plot()

The plot() function is being called to do the task of plotting the graph given the points initialized by the plotdata class. This function opens a separate window where the 2-dimensional graph is drawn.

The plot() function is called in the following manner;
plot(xAxis, yAxis, colour, label);

where xAxis and yAxis are the plotdata variables which contain the data for the horizontal axis and the vertical axis respectively. The variable colour pertains to the color of the graph and the label is the string that contains the header of the graph window. The last two variables can be excluded in the plotting depending on the desires of the user. When either of the variables, or both variables, is not called, the default values are used.

The audio waveform display program calls the function plot() after the program has already read through the loaded .wav file. The plot() function as called in the following manner;

plot (x, y, NULL);

A NULL argument is called to replace the label and the colour variables in order to obtain a default setting. When the plot window appears, the default color chosen by the program is a deep red with a header “2D Plot” on its title bar. The interface is a static window where the user has limited functions to manipulate with, similar to that of an image. However, when another .wav file is loaded, the previous window does not disappear and the new input .wav file is plotted in a separate window.

3.1.2.3. addMark()

The plotting of the graph itself was done through the function addMark(). Through this function, the points in the graph appear as small circles instead of a scatter of points connected by a line. The function marks the position of the coordinates in the plane. This function is called as coded below;

addMark(x, y, xCoord, yCoord);
The variables xCoord and yCoord pertain to the x and y coordinates that are to be plotted in the graph. In the program, the addMark() plot function is called as:

```
addMark(x, y, j, pdmdata[i]);
```

where the pdmdata[i] pertains to the amplitude of the samples.

### 3.1.2.4. axesBotLeft(); axesTopRight()

Koolplot has a built-in function that adjusts the size of the window highly dependent on the range defined by the user. The library controls two parts of the window, the bottom left and the top right [22]. Shown below is the implementation of the code for the function:

```
axesBotLeft(xData, yData, minX, minY);
// enables extending the bottom left corner of the axes (downwards and left-wards, away from the graph)

axesTopRight(xData, yData, maxX, maxY);
// enables extending the top right corner of the axes (upwards and right-wards, away from the graph)
```
4. Conclusion and Recommendations

A working Graphical Audio Waveform Display that displays a characteristic waveform for a given wavefile was created. The said program allowed multiple window display of several audio files. The code for data extraction, from the previous projects, was discarded to come up with a new code that was easier to understand and reuse in other programs. Two classes, WAV_IN and WAV_OUT were derived from source codes and made to match the programs functions. Additionally, the graphing algorithm was changed and developed with the help of koolplot, an open source library specifically used for graphing in C++.

However, the necessity to change the whole program as to allow the multiple window display that was aimed for this project limited the operability of the old graphic user interface. The interface for the program is currently limited to loading the input in the command prompt.

The group recommends that a Graphic User Interface that would nest the windows created by this program be made for easier usage. Also, the group has been limited to the graphing of WAV files only, attempts to create an input algorithm to load MP3 and other audio files may be done. Finally, customizability of the resulting graphs may be offered to future reviewers of this program.
Appendix 1. Software Source Codes

A.1 driver.cpp

// driver.cpp
// Example program that uses the WAV_IN and WAV_OUT classes
// written by Dr. Fred DePiero.
// t a y l o r@msoe.edu, 6-26-2002
// adapted for audio waveform display

#include "koolplot.h"
#include <iostream>
#include "wav_in.h"
#include "wav_out.h"
#include "wav_in.cpp"
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv)
{
    plotdata x,y;
    if(argc!=3) {
        std::cerr<<" 
This program loads an audio wave file\n"
            << " and inputs a characteristic waveform. \n"
            << " Edited the utility code written by Dr. Fred DePiero\n";
    } else {

        WAV_IN infile(argv[1]);
        int pcmdata(infile.get_num_samples());
        double sampleRate = infile.get_sample_rate_hz();
        double data;
        unsigned int bitsPerSample = infile.get_bits_per_sample();
        unsigned int channels = infile.get_num_channels();

        int i, j;

        i = 0;
        j = 1;

        while(infile.more_data_available()) {
            data = infile.read_current_input();
            pcmdata[i] = int(data);
            printf("%d",j);
            printf("%d\n",pcmdata[i]);
        }
    }
}
addMark(x, y, j, pcmdata[i]);
i = i + 1;
j = j + 1;
}

plot(x, y, NULL);

system("pause");
return 0;
}

A.2. wav_out.cpp

/******************** FILE : wav_out.cpp ********************/
/****** NOTICE: LIMITATIONS ON USE AND DISTRIBUTION ********/

This software is provided on an as-is basis, it may be distributed in an unlimited fashion without charge provided that it is used for scholarly purposes - it is not for commercial use or profit. This notice must remain unaltered.

Software by Dr Fred DePiero - CalPoly State University

/******************** END OF NOTICE ********************/
/ enlarge buffer
if(g_num_osamp>=g_max_osamp)
{
    g_max_osamp *= 2;
    tmp = new double [g_max_osamp];
    if(tmp==NULL){ printf("cant realloc in WAV_OUT\n"); exit(-1); }

    // copy over
    for(i=0;i<g_num_osamp;i++) tmp[i] = g_wdata_out[i];
    for(i=g_num_osamp;i<g_max_osamp;i++) tmp[i] = 0.0;

    // swap buffers
    delete g_wdata_out;
    g_wdata_out = tmp;
}

// buffer input data
  g_wdata_out[g_num_osamp++] = ooo;
return 0;

/* int WAV_OUT::write(double ooo) */

int WAV_OUT::save_wave_file(char *fname)
{
    FILE *fw;
    unsigned int wstat;
    int i;
    char obuff[80];

    WAV_HDR *wav;
    CHUNK_HDR *chk;
    char *wbuff;
    int wbuff_len;

    short int *uptr;
    double ttt;
    double max_uuu = (65536.0 / 2.0) - 1.0;
    double min_uuu = -(65536.0 / 2.0);
    unsigned char *cptr;
    double max_ccc = 256.0;
    double min_ccc = 0.0;

    if(g_num_osamp<=0) printf("No new data was written to output. \n");

    // allocate wav header

wav = new WAV_HDR;
chk = new CHUNK_HDR;
if(wav==NULL){ printf("Can't new headers\n"); exit(-1); }
if(chk==NULL){ printf("Can't new headers\n"); exit(-1); }

/* allocate new data buffers */
wbuff_len = g_num_osamp * bits_per_sample / 8;
wbuff = new char [wbuff_len];
if(wbuff==NULL){ printf("Cannot allocate buffer. \n"); exit(-1); }

// setup wav header
sprintf(obuff,"RIFF");
for(i=0;i<4;i++) wav->rID[i] = obuff[i];

sprintf(obuff,"WAVE");
for(i=0;i<4;i++) wav->wID[i] = obuff[i];

sprintf(obuff,"fmt ");
for(i=0;i<4;i++) wav->fId[i] = obuff[i];

wav->nBitsPerSample = bits_per_sample;
wav->nSamplesPerSec = (int) fs_hz;
wav->nAvgBytesPerSec = (int) fs_hz;
wav->nAvgBytesPerSec *= bits_per_sample / 8;
wav->nChannels = num_ch;
wav->pcm_header_len = 16;
wav->wFormatTag = 1;
wav->rLen = sizeof(WAV_HDR) + sizeof(CHUNK_HDR) + wbuff_len;
wav->nBlockAlign = num_ch * bits_per_sample / 8;

// setup chunk header
sprintf(obuff,"data");
for(i=0;i<4;i++) chk->dId[i] = obuff[i];

chk->dLen = wbuff_len;

// convert data
if(bits_per_sample == 16){
  uptr = (short *) wbuff;
  for(i=0;i<g_num_osamp;i++){
    ttt = g_wdata_out[i];
    if(ttt>max_uuu) ttt = max_uuu;
    if(ttt<min_uuu) ttt = min_uuu;
    uptr[i] = (short int) ttt;
  }
else if(bits_per_sample == 8){
    cptr = (unsigned char *) wbuff;
    for(i=0;i<g_num_osamp;i++){
        ttt = g_wdata_out[i];
        if(ttt>max_ccc) ttt = max_ccc;
        if(ttt<min_ccc) ttt = min_ccc;
        cptr[i] = (unsigned char) ttt;
    }
}
else{ printf("bunk bits_per_sample\n"); exit(-1); }
/* open wav file */
fw = fopen(fname,"wb");
if(fw==NULL){ printf("Cannot open WAV file\n"); exit(-1); }

/* write riff/wav header */
wstat = fwrite((void *)wav,sizeof(WAV_HDR),(size_t)1,fw);
if(wstat!=1){ printf("Cannot write WAV file\n"); exit(-1); }

/* write chunk header */
wstat = fwrite((void *)chk,sizeof(CHUNK_HDR),(size_t)1,fw);
if(wstat!=1){ printf("Cannot write chunk header\n"); exit(-1); }

/* write data */
wstat = fwrite((void *)wbuff,wbuff_len,(size_t)1,fw);
if(wstat!=1){ printf("Cannot write wbuffer\n"); exit(-1); }

printf("\nSaved WAV File: %s\n",fname);
printf(" Sample Rate = %1.0lf (Hz)\n",fs_hz);
printf(" Number of Samples = %ld\n",g_num_osamp);
printf(" Bits Per Sample = %d\n",bits_per_sample);
printf(" Number of Channels = %d\n",num_ch);

// reset output stream index
  g_num_osamp = 0;

if(wbuff!=NULL) delete wbuff;
if(wav!=NULL) delete wav;
if(chk!=NULL) delete chk;
fclose(fw);
return 0;

}/* int WAV_OUT::save_wave_file(char *fname) */
WAV_OUT::WAV_OUT(double _fs_hz, int _bits_per_sample, int _num_ch) {
    
    fs_hz = _fs_hz;
    bits_per_sample = _bits_per_sample;
    num_ch = _num_ch;

    g_wdata_out = NULL;
    g_num_osamp = 0;
    g_max_osamp = 0;

    return;

    /* WAV_OUT::WAV_OUT(), */
}

WAV_OUT::WAV_OUT(WAV_IN *wav_in) {
    fs_hz = wav_in->fs_hz;
    bits_per_sample = wav_in->bits_per_sample;
    num_ch = wav_in->num_ch;
    g_wdata_out = NULL;
    g_num_osamp = 0;
    g_max_osamp = 0;

    return;

    /* WAV_OUT::WAV_OUT(), */

WAV_OUT::~WAV_OUT(){ }

A.3. wav_in.cpp

/******************** FILE : wav_in.cpp ********************/
/****** NOTICE: LIMITATIONS ON USE AND DISTRIBUTION **********/

This software is provided on an as-is basis, it may be distributed in an unlimited fashion without charge provided that it is used for scholarly purposes - it is not for commercial use or profit. This notice must remain unaltered.

Software by Dr Fred DePiero - CalPoly State University

END OF NOTICE ********************/

#include <cstdio>
#include <cstdlib>
#include <cmath>
#include "f_err.h"
#include "f_ptch.h"
```c
#include "wav_def.h"
#include "wav_in.h"

long int WAV_IN::get_num_samples(){ return g_max_isamp; }
int WAV_IN::get_num_channels(){ return num_ch; }
int WAV_IN::get_bits_per_sample(){ return bits_per_sample; }
double WAV_IN::get_sample_rate_hz(){ return fs_hz; }

int WAV_IN::more_data_available()
{
    if(g_num_isamp>=g_max_isamp) return 0;
    return 1;
}

double WAV_IN::read_current_input()
{
    if( (g_wdata_in==NULL) || (g_max_isamp<=0) || (g_num_isamp<0) )
    {
        printf("The file input was not loaded.\n");
        exit(1);
    }
    if(g_num_isamp>=g_max_isamp)
    {
        printf("Attempting to read past end of input buffer.\n");
        exit(1);
    }
    //printf("%f\n", g_wdata_in[g_num_isamp++]);
    return( g_wdata_in[g_num_isamp++] );
}

WAV_IN::WAV_IN(char *file_name)
{
    int i;
    FILE *fw;
    unsigned int wstat;
    char obuff[80];

    WAV_HDR *wav;
    CHUNK_HDR *chk;
    short int *uptr;
    unsigned char *cptr;
    int sflag;
    long int rmore;
    double ace;
    char *wbuff;
```
int wbuff_len;

// set defaults
g_wdata_in = NULL;
g_num_isamp = 0;
g_max_isamp = 0;

// allocate wav header
wav = new WAV_HDR;
chk = new CHUNK_HDR;
if(wav==NULL){ printf("Cannot open new headers\n"); exit(-1); }
if(chk==NULL){ printf("Cannot open new headers\n"); exit(-1); }

/* open wav file */
fw = fopen(file_name,"rb");
if(fw==NULL){ printf("Cannot open WAV file\n"); exit(-1); }

/* read riff/wav header */
wstat = fread((void *)wav,sizeof(WAV_HDR),(size_t)1,fw);
if(wstat!=1){ printf("Cannot read WAV header\n"); exit(-1); }

// check format of header
for(i=0;i<4;i++) obuff[i] = wav->rID[i];
obuff[4] = 0;
if(strcmp(obuff,"RIFF")!=0){ printf("Bad RIFF format\n"); exit(-1); }

for(i=0;i<4;i++) obuff[i] = wav->wID[i];
obuff[4] = 0;
if(strcmp(obuff,"WAVE")!=0){ printf("Bad WAVE format\n"); exit(-1); }

for(i=0;i<3;i++) obuff[i] = wav->fId[i];
obuff[3] = 0;
if(strcmp(obuff,"fmt")!=0){ printf("Bad FMT format\n"); exit(-1); }

if(wav->wFormatTag!=1){ printf("Bad WAV wFormatTag\n"); exit(-1); }

if( (wav->nBitsPerSample != 16) && (wav->nBitsPerSample != 8) ){
    printf("Bad WAV nBitsPerSample\n"); exit(-1); }

// skip over any remaining portion of wav header
rmore = wav->pcm_header_len - (sizeof(WAV_HDR) - 20);
wstat = fseek(fw,rmore,SEEK_CUR);
if(wstat!=0){ printf("Cannot seek \n"); exit(-1); }

// read chunks until a 'data' chunk is found
sflag = 1;
while(sflag!=0){

    // check attempts
    if(sflag>10){ printf("There are too many chunks\n"); exit(-1); }

    // read chunk header
    wstat = fread((void *)chk,sizeof(CHUNK_HDR),(size_t)1,fw);
    if(wstat!=1){ printf("Chunk cannot seek.\n"); exit(-1); }

    // check chunk type
    for(i=0;i<4;i++) obuff[i] = chk->dId[i];
    obuff[4] = 0;
    if(strcmp(obuff,"data")==0) break;

    // skip over chunk
    sflag++;
    wstat = fseek(fw,chk->dLen,SEEK_CUR);
    if(wstat!=0){ printf("Cannot seek.\n"); exit(-1); }
}

/* find length of remaining data */
wbuff_len = chk->dLen;

/* find number of samples */
g_max_isamp = chk->dLen;
g_max_isamp /= wav->nBitsPerSample / 8;

/* allocate new buffers */
wbuff = new char [wbuff_len];
if(wbuff==NULL){ printf("Cannot allocate new buffer. \n"); exit(-1); }

//if(g_wdata_in!=NULL) delete g_wdata_in;
g_wdata_in = new double [g_max_isamp];
if(g_wdata_in==NULL){ printf("Cannot allocate new buffer. \n"); exit(-1); }

/* read signal data */
wstat = fread((void *)wbuff,wbuff_len,(size_t)1,fw);
if(wstat!=1){ printf("Cannot read wbuffer. \n"); exit(-1); }

// reset buffer stream index
g_num_isamp = 0;

if(wbuff!=NULL) delete wbuff;
if(wav!=NULL) delete wav;
if(chk!=NULL) delete chk;
fclose(fw);
return;

/* WAV_IN::WAV_IN() */
WAV_IN::~WAV_IN(){ }

A.4. common.h

/*
This program is distributed under the terms of the 'MIT license'. The text
of this licence follows...

Copyright (c) 2004-2009 J.D.Medhurst (a.k.a. Tixy)

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN
THE SOFTWARE.
*/

/**
@file
@brief Type definitions and helper macros which aren't part of Standard C++
@version 2007-01-17
- Fixed definition of <CODE>offsetof</CODE> to work on GGC 4.
- Added global placement new operator for constructing an object at a specified address.
*/
/*

#ifndef COMMON_H
#define COMMON_H
#if _DEBUG
#undef DEBUG
#define DEBUG /**< Defined when compiling code for debugging */
#endif
#if defined(_MSC_VER)
// Compiling for Microsoft Visual C++...
#define BREAKPOINT { _asm int 3 } /**< Invoke debugger */
#define ASSERT(c) {if(!(c)) BREAKPOINT;} /**< Assert that expression 'c'
is true */
typedef __int64 longlong;
typedef unsigned __int64 ulonglong;

// disable annoying warnings from MSVC...
#pragma warning( disable : 4244 ) /* conversion from x to y, possible loss of data */
#pragma warning( disable : 4514 ) /* unreferenced inline function has been removed */
#pragma warning( disable : 4146 ) /* unary minus operator applied to unsigned type, result still unsigned */
#pragma warning( disable : 4710 ) /* function x not inlined */
#pragma warning( disable : 4355 ) /* 'this': used in base member initializer list */
#pragma warning( disable : 4512 ) /* assignment operator could not be generated */
#pragma warning( disable : 4800 ) /* forcing value to bool 'true' or 'false' (performance warning) */
#else
#define BREAKPOINT /**< Invoke debugger */
#endif

#elif defined(__EPOC32__) || defined(__WINS__)
// Compiling for Symbian OS...
#define COMPILE_FOR_SYMBIAN
#include <e32std.h>
#if !defined(__BREAKPOINT)
#if defined(__WINS__)
#if defined(__WINSCW__)}
#define BREAKPOINT { _asm byte 0xcc } /**< Invoke debugger */
#else
#define BREAKPOINT { _asm int 3 } /**< Invoke debugger */
#endif
#else
#define BREAKPOINT /**< Invoke debugger */
#endif
#endif
#define BREAKPOINT {__BREAKPOINT()} /**< Invoke debugger */
#endif

*/
#undef ASSERT
#define ASSERT(c) {if(!(c)) BREAKPOINT;} /**< Assert that expression 'c' is true */

#if !defined(DEBUG) && defined(_DEBUG)
#define DEBUG /**< Defined when compiling code for debugging */
#endif

typedef TInt64 longlong;
typedef TUint64 ulonglong;

#else
// Compiling for unknown system...
#define BREAKPOINT /**< Invoke debugger */

extern int AssertFailed(const char* file, int line);
#undef ASSERT
#define ASSERT(c) (void)((c)||AssertFailed(__FILE__,__LINE__)) /**< Assert that expression 'c' is true */

typedef long long longlong;
typedef unsigned long long ulonglong;
#endif

#ifdef DEBUG
#define ASSERT_DEBUG(c) ASSERT(c) /**< Assert that expression 'c' is true (when compiled for debugging)*/
#else
#define ASSERT_DEBUG(c) /**< Assert that expression 'c' is true (when compiled for debugging)*/
#endif

#ifndef ASSERT_COMPILE
/** Assert, at compile time, that expression 'c' is true. */
#define ASSERT_COMPILE(c) void assert_compile(int assert_compile[(c)?1:-1])
#endif

@defgroup integers Common - Basic Integer Types.

These definitions will need to be modified on systems where 'char', 'short' and 'int' have sizes different from 8, 16 and 32 bits.
Note, 'int' is assumed to be in 2s complement format and at least 32 bits in size.
@
/*
typedef unsigned char uint8_t; /**< An 8 bit unsigned integer */
typedef unsigned short uint16_t; /**< An 16 bit unsigned integer */
typedef unsigned int uint32_t; /**< An 32 bit unsigned integer */
typedef ulonglong uint64_t; /**< An 64 bit unsigned integer */
typedef signed char int8_t; /**< An 8 bit signed integer (2s complement) */
typedef signed short int16_t; /**< An 16 bit signed integer (2s complement) */
typedef signed int int32_t; /**< An 32 bit signed integer (2s complement) */
typedef longlong int64_t; /**< An 64 bit signed integer (2s complement) */
typedef int intptr_t; /**< An signed integer of the same size as a pointer type */
typedef unsigned int uintptr_t; /**< An unsigned integer of the same size as a pointer type */
typedef int64_t intmax_t; /**< Largest signed integer type */
typedef uint64_t uintmax_t; /**< Largest unsigned integer type */
typedef intptr_t size_t; /**< A size of an object or memory region */
typedef intptr_t ptrdiff_t; /**< A signed integer which can hold the different between two pointer */

/** @} */ // End of group

ASSERT_COMPILE(sizeof(uintptr_t)==sizeof(void*));
ASSERT_COMPILE(sizeof(intptr_t)==sizeof(void*));
#if __GNUC__<4
/** Calculate address offset of member within a type. */
#define offsetof(type/member) ((size_t)((type*)256)->member)-256
#else
#define offsetof(type/member) __builtin_offsetof(type,member)
#endif
#if defined(__GNUC__) && defined(_ARM)
/** Used to stop GCC "warning: control reaches end of non-void function" in __naked__ functions. */
#define dummy_return(type) register type _r0 asm("r0"); asm("": "r"(_r0)); return _r0
#endif
#ifndef COMPILE_FOR_SYMBIAN

/**

*/
#define define dummy_return(type) register type _r0 asm("r0"); asm("": "r"(_r0)); return _r0
#endif
#ifndef COMPILE_FOR_SYMBIAN
/** Global placement new operator for constructing an object at a specified address */
inline void* operator new(size_t, void* ptr) throw()
    { return ptr; }

/** Global placement delete operator. */
inline void operator delete(void*, void*) throw()
    { }
#endif // !COMPILE_FOR_SYMBIAN
#endif

A.5. f_err.h

#ifndef FERR_H
#define FERR_H
#define thret_gerr(n,s) { printf("%s (%d)\n",s,n); exit(n); }
#define threx_gerr(n,s) { printf("%s (%d)\n",s,n); exit(n); }
#define threv_gerr(n,s) { printf("%s (%d)\n",s,n); exit(n); }
#define throw_gerr(n,s) { printf("%s (%d)\n",s,n); exit(n); }
#define reton_gerr()
#define revon_gerr()
#define clear_gerr()
#define print_gerr()
#define catch_gerr()  0
#endif

A.6. f_ptch.h

#ifndef F_PTCH_HH
#define F_PTCH_HH
#include <string.h>
#include <memory.h>
#define PI (3.14159265359)
#define bcopy(src,dest,sz) memcpy(dest,src,(size_t)sz)
#endif

A.7. wav_def.h

/******************** FILE : wav_def.h ********************/
/***** NOTICE: LIMITATIONS ON USE AND DISTRIBUTION *******/

This software is provided on an as-is basis, it may be
distributed in an unlimited fashion without charge provided
that it is used for scholarly purposes - it is not for
commercial use or profit. This notice must remain unaltered.
typedef struct {
    char rID[4];            // 'RIFF'
    long int rLen;

    char wID[4];            // 'WAVE'

    char fId[4];            // 'fmt'
    long int pcm_header_len;   // varies...
    short int wFormatTag;
    short int nChannels;      // 1,2 for stereo data is (l,r) pairs
    long int nSamplesPerSec;
    long int nAvgBytesPerSec;
    short int nBlockAlign;
    short int nBitsPerSample;
} WAV_HDR;

typedef struct {
    char dId[4];            // 'data' or 'fact'
    long int dLen;
    // unsigned char *data;
} CHUNK_HDR;

A.8. wav_in.h

 ifndef INCLUDE_WAV_IN
 define INCLUDE_WAV_IN

 class WAV_OUT;

class WAV_IN
{
public:

    WAV_IN(char *wav_file_name);
    ~WAV_IN();
    // routine for reading one sample from a (previously loaded) wave file
    // returns current sample as a double
    double read_current_input();

    // determines end-of-file condition, returns 1==true if more data ready
    int more_data_available();

    // returns number of samples in file
    long int get_num_samples();

    // reports number of channels (1==mono, 2==stereo)
    int get_num_channels();

    // reports the number of bits in each sample
    int get_bits_per_sample();

    // reports sample rate in Hz
    double get_sample_rate_hz();

protected:

    double fs_hz;
    int bits_per_sample;
    int num_ch;

    double *g_wdata_in;
    int g_num_isamp;
    long int g_max_isamp;

    friend class WAV_OUT;
};
#endif

A.9. wav_out.h
/******************* FILE : wav_out.h *******************/
/****** NOTICE: LIMITATIONS ON USE AND DISTRIBUTION *******

This software is provided on an as-is basis, it may be
distributed in an unlimited fashion without charge provided
that it is used for scholarly purposes - it is not for
#ifndef INCLUDE_WAV_OUT
#define INCLUDE_WAV_OUT

class WAV_IN;
class WAV_OUT
{

public:

    // create a new wav_out with given parameters
    // note: soundcards typically support a limited range of values!
    //       hence the next constructor is safer: WAV_OUT(WAV_IN *wav);
    WAV_OUT(double fs_hz,int bits_per_sample,int num_ch);

    // create a wav_out with the same parameters as a given wav_in
    WAV_OUT(WAV_IN *wav_in);

    ~WAV_OUT();

    // routine for writing one output sample
    // samples are stored in a buffer, until save_wave_file() is called
    // returns 0 on success
    int write_current_output(double ooo);

    // routine for saving a wave file.
    // returns 0 on success, negative value on error
    int save_wave_file(char *wav_file_name);

protected:

    double fs_hz;
    int bits_per_sample;
    int num_ch;

    double *g_wdata_out;
    int g_num_osamp;
    long int g_max_osamp;
    friend class WAV_IN;

} #endif
References


