MP9 – Audio processing

Due Wednesday November 12th, 10 pm

Your task this week is to implement 4 functions to manipulate .wav files. The four functions are **little_endian_2(), little_endian_4(), read_file(),** and **sloop()** **little_endian_2()** and **little_endian_4()** are functions that will read chunks of data from a file and help you complete **read_file(). read_file()** will read in a wav file, fill out a wav file struct which has been designed for you, and print out the characteristics of the wav file. Finally, **sloop()** takes a wav file and outputs another wav file which loops from point a to point b n number of times.
The Pieces
In the MP9 folder, you will find the following 3 programs.
prog9.c - This file contains the function definitions for each of the 4 functions listed above. You will edit this file by adding in the functionality of the given functions.
prog9.h - This file contains the function prototypes for each of the 4 functions as well as the struct definition. Do not edit this file.
main.c - This file will allow the user to choose between 2 data manipulation functions to use (the second one is if you do the extra credit). It also takes the input from the command line and feeds it to the appropriate function based on the input. Do not edit this file.
Below are the function prototypes for the 4 functions that you will be implementing. We recommend implementing them in this order.

```
int little_endian_2(FILE *fptr);
int little_endian_4(FILE *fptr);
WAV *read_file(char *wavfile);
void sound_bite(WAV *inwav, char *outfile, double a, double b, int n);
```

Details

A .wav file is a file format standard for storing an audio bitstream on PCs. It contains several bytes of header information that details many things about the bitstream including the number of channels, the sample rate, and the overall size of the file etc. After the header, the data is stored in a manner consistent with the header description. Not all wav file headers are the same but they follow a similar format and differ usually only on the amount of bytes dedicated to a particular characteristic field.
HERE (https://ccrma.stanford.edu/courses/422/projects/WaveFormat/) is a link which provides a detailed description of the WAV file header. This link is divided into 3 main parts which say essentially the same thing in 3 different ways. The first section gives a general breakdown of the structure of a wav file. The second section gives details about the wav file header components and how those numbers are calculated. The third section gives a detailed breakdown of an example header from a wav file. There is also a "Notes" section which will also be useful to read. Focus mainly on the first and third sections as they give

information that is most useful for this lab.

Endianess

You will probably notice that in the description of the header in the first section, there is a column for "endian". This essentially is describing in which order the parts of a particular piece of information are stored. In the LC-3, each memory address was 16 bits wide and every piece of information that we used was also 16 bits wide. Thus, each memory or register access gave us the entire piece of data that we wanted. However, if we wanted to represent a 32 or 64 bit piece of data, and our memory was still only 16 bits wide, we would have to break the data into 2 or 4 separate pieces respectively. As explained before, endianess tells which part of those pieces are stored first in memory.
For example, if we want to store the 32 bit number 0x12345678 into LC-3 memory, then we will need to use two memory locations. Let's say we will store the number into 0x4000 and 0x4001. If we use a big endian scheme, then 0x1234 will be stored at 0x4000 and 0x5678 will be stored at 0x4001. Conversely, if we use a little endian scheme, then the little end will go first, that is to say 0x5678 will be stored at 0x4000 and 0x1234 will be stored at 0x4001. Now, let us explore a quick example from the example header file. The four bytes of data associated with the SampleRate are (in the order given in the file) 22 56 00 00. In big endian notation the byte rate would be 0x22560000 (576,061,440 times a second. Intuition here warns us that we probably don't want to be sampling our signal half a billion times a second). However, in little endian notation we have 0x00005622 (22,050 times a second sounds much more reasonable).

little_endian_2()

This function will be used to more easily read data from the wav file. The "little_endian" part of the function name signifies that the data that we are going to read is going to be stored in little endian notation. The "2" indicates the number of bytes that we will read. Inside this function, we will use fscanf to read 2 bytes worth of data, reorganize it into big endian notation and return that value. An example is if we were to call little_endian_2() where the file pointer was pointing to 7F 1C, the number that would be returned from this function call would be 7,295 (or x1C7F in hexadecimal).

little_endian_4()

This function is exactly the same as little_endian_2() except that this function will use fscanf to read 4 bytes worth of data,reorganize it into big endian notation and return that value.

read_file()

For the third function, **read_file()**, you will need to read through the header of a wav file, fill out a wav file struct, and print out to the the screen all of the relevant information.
Here is the code for the wav file struct that we give you:

```
struct wav_t{
  char                    RIFF[4];
  int                     ChunkSize;
  char                    WAVE[4];
  char                    fmt[4];
  int                     Subchunk1Size;
  short int               AudioFormat;
  short int               NumChan;
  int                     SamplesPerSec;
  int                     bytesPerSec;
  short int               blockAlign;
  short int               bitsPerSample;
  char                   *extra;
  char                    Subchunk2ID[4];
  int                     Subchunk2Size;
  short int              *data;
};
```

You will notice that there are 4 char arrays of size 4. These will store letters from the wav file and denote breaks in the header structure. The different segments are referred to as "subchunks". Also notice that there are two pointers in the struct, namely "extra" and "data". These are both pointers because they will serve as arrays that will be dynamically allocated. Like its name suggests, "extra" holds extra bytes of information that are not necessary to this lab, if they exist in the wav file at all.

malloc
malloc is a way of creating an element like an array or struct or even something as small as an int or char, and having it be created and stored somewhere other than on the stack. This characteristic allows whatever was malloc'd to persist longer than the function which called it. If you will recall, all local variables created are pushed onto the stack and whenever the function returns, those values are lost. With malloc, the values created will continue to exist after the function returns.
The code for malloc'ing a WAV file struct is as follows:
`WAV* wav_ptr = (WAV *)malloc(sizeof(WAV));`
This is essentially saying, "Set aside enough permanent space to fit a WAV file

struct. From here on out, I will refer to that space using the WAV pointer named wav_ptr."
The other two malloc's that you will need to do will look just like this:

```
wav_ptr->extra=(char*)malloc((the number of elements in the "extra" array)*sizeof(char));
AND
wav_ptr->data=(short int*)malloc((the number of elements in the "data" array)*sizeof(short int));
```

After these two calls, data and extra can be accessed as normal arrays: wav_ptr->extra[i] and wav_ptr->data[0] etc...
The number of char elements that should be malloced to "extra" can be calculated using the following formula: (Subchunk1Size-16). The 16 comes from the number of bytes that would be contained in the Subchunk1 if "extra" was empty. The "data" array will have as many elements as there are samples in the wav file. This number can be determined by taking the total number of bytes of data, multiplying it by 8 (number of bits in a byte) and dividing by the bitsPerSample. Overall, there will be 3 malloc calls that you need to make. The first is one to actually create the wav struct. This will happen right at the beginning of read_file() and this is the WAV* that you will return. The second and third will happen after that as you read through the header file and will be for "extra" and "data" as we just discussed.

Back to read_file()

When printing out the information in the header, use the naming convention provided in the first section in the WAV file link above. For example, if we have a header file that contains the following bytes as its header:

```
52 49 46 46 24 08 00 00 57 41 56 45 66 6d 74 20 10 00 00 00 01 00
02 00 22 56 00 00 88 58 01 00 04 00 10 00 64 61 74 61 00 08 00 00
```

We will expect an output of the following after read_file() has been called:

```
ChunkID: RIFF
ChunkSize: 2084
Format: WAVE
Subchunk1ID: fmt
Subchunk1Size: 16
AudioFormat: 1
NumChannels: 2
SampleRate: 22050
ByteRate: 88200
BlockAlign: 4
BitsPerSample: 16
Subchunk2ID: data
Subchunk2Size: 2048
```

See the third section in the wav file link for a more detailed breakdown of how these values were calculated.
Note that the data samples are also stored little endian. This will mean that you

should use little_endian_2() to read out all of the data values and store them into the wav_ptr->data[] array. Also keep in mind that there will be 2 channels sampled which means that chars will alternate between which speaker (right or left) their sound goes to. For example, the pattern will be as follows: for one time sample there will be 2 chars for the left speaker and then 2 chars for the right speaker and then for the next time sample there will also be 2 for the left then 2 for the right making a total of 4 bytes of data for each time step.

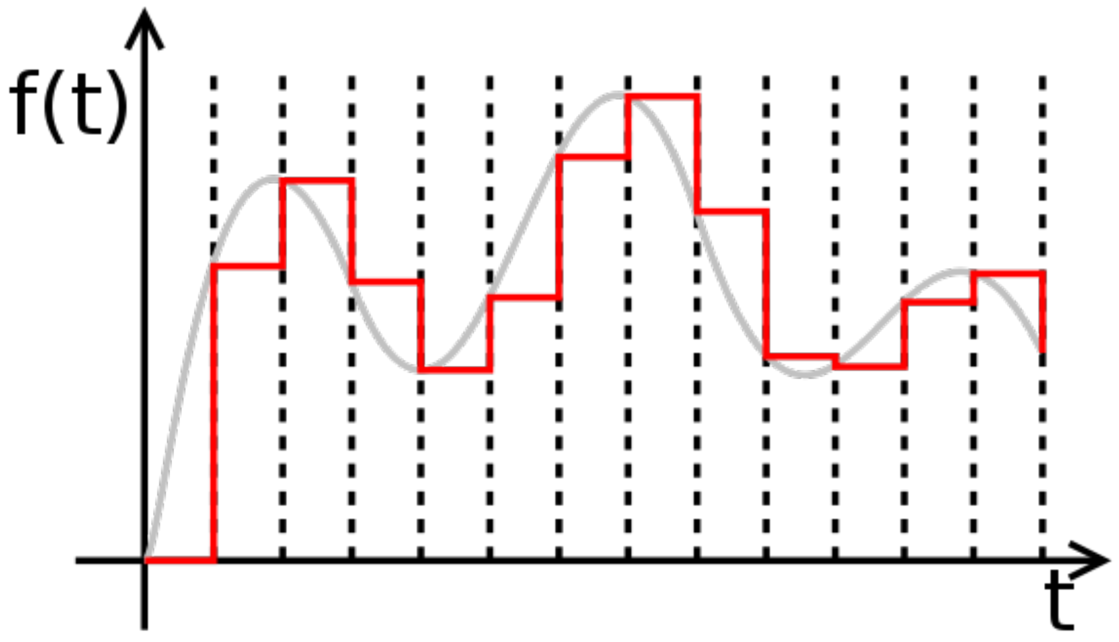
sloop()

For the sloop() function, in essence, all we are doing is looping the given file n number of times. By doing this, we generate a new wav file which has n times more length and has samples of the original wav file repeated n times. In sloop, you will be passed 5 values. The first is a pointer to a wav struct that has already been read from the file, which we are repeating. The second is the name of the new file that we make and the last is the number of times we want to repeat the file. Third and fourth are the starting and stopping times instants of time. Essentially all that sloop() does is copy the entire header file from the original wav file to the new file changing the number of data values. And copy the data of the original wav from start to stop to the output wav file n number of times.
If you are interested in working with other wav files than the ones we have provided, http://audio.online-convert.com/convert-to-wav is a website which will convert mp3 files into a wav file format.

Challenge for MP9

For this challenge, we will be implementing a lo pass filter. However, before we talk about the lo pass filter function, let's discuss what data is actually stored in the wav file and what exactly it means.

Sound is an oscillation of pressure transmitted through a solid, liquid, or gas. When you hear different volumes and pitches of sound all that is happening is that each sound wave varies in energy for the volume (larger energy waves, the louder the sound), or distance between sound waves which adjusts the pitch, (smaller distances between waves leads to higher pitched sound). Sound waves in nature are continuous, this means they have an infinite amount of detail that you could store for even the shortest sound. This makes them very difficult to record perfectly and so what computers do is sample the analog signal many thousands of times per second and get discrete values back. These samples are then played back and our ears interpret them as the continuous waves. In the image below, the gray line represents the actual continuous audio and the red line represents the samples taken by the computer at a particular sample rate. These samples are the data which is stored in the wav file for us to play back and manipulate.



For the lo pass filter function, we will be using the following mathematical function.

```
out(n) = (a1 * in(n)) + (a2 * in(n-1)) + (a3 * in(n-2)) - (b1 * out(n-1)) - (b2 * out(n-2))
```

Here, out(n) is the current time step that we are about to output and in(n) is the current value from the "data" array. out(n-1) and in(n-1) correspond to the previous output and input values. out(n-2) and in(n-2) correspond to the input and output values from 2 time steps previous. Keep in mind that there are two

channels when accessing your "data" array and that the previous sample for a speaker will not be in an adjacent array location.
The formulas for computing the coefficients are as follows:

```
r = resolution       // we will use 0.5 to test, feel free to change this for different results.
                     // sqrt(2) to 0.1 works best.

f = cutoff frequency //for lo pass, we cut off everything above f.
                     //1000 works well for lo pass.
```

lo_pass()

```
c = 1.0 / tan( pi * f / sample_rate);

a1 = 1.0 / ( 1.0 + r * c + c * c);
a2 = 2.0 * a1;
a3 = a1;
b1 = 2.0 * ( 1.0 - c*c) * a1;
b2 = ( 1.0 - r * c + c * c) * a1;
```

Data types are an extremely important consideration for this lab. The values stored in the wav files are not ASCII characters representing the values. Rather, they are simply stored as individual bytes and thus must be treated as such. Something else to keep in mind is that for the majority of wav files, the data values are 16 bits (the datatype is a *short int*). Thus, in order to read in one whole value, you should read in two char values and compose them in the appropriate way to form a 16 bit value. This can be done using little_endian_2(). Similarly, when writing back to the file, take this 16 bit value and output the lower 8 bits as a char and then the upper 8 bits as a char since it is little endian.

Building and Testing

The implemented functions will be associated with the following commands:
1 : **sloop()**
2 : **lo_pass()**
To compile your MP, use the following command:
`gcc -Wall -lm main.c prog9.c -o prog9`
To run your code, you will need to provide multiple inputs on the command line. The first is the number (1 or 2) of the function that you want to test. The second will be the input file that you are primarily reading. Based on which function that you want to test, the subsequent inputs will be different. Here are command examples of each of the 4 functions.

sloop ()

```
./prog9 1 <inputsong.wav> <outputsong.wav> n a b  //the input from a to b will be repeated n times
```

lo_pass() EXTRA CREDIT

```
./prog9 2 <inputsong.wav> <outputsong.wav> 1000   //the cutoff frequency will be 1000
```

We have given you omg.wav. In order to test your implementation, after compiling, run the following command:

```
./prog9 1 piano.wav out.wav 5 4 8//4 to 8 seconds of piano.wav will be looped 5 times.
```

Here is the output that should be printed out to the console after this command:

```
RIFF: RIFF
ChunkSize: 22735910
WAVE: WAVE
fmt: fmt
Subchunk1Size: 18
AudioFormat: 1
NumChan: 2
SamplesPerSec: 44100
bytesPerSec: 176400
blockAlign: 4
bitsPerSample: 16
Subchunk2ID: data
Subchunk2Size: 22735872
```

Grading Rubric

Functionality (70%)
(10%) - Correctly implements the low pass filter function little_endian_2()
(10%) - Correctly implements the low pass filter function little_endian_4()
(25%) - Correctly implements the wav file input function read_file()
(25%) - Correctly implements the looping function sloop()
Style (15%)
(5%) - Compilation generates no warnings.
(10%) - Indentation and variables names are appropriate and meaningful
Comments, Clarity and Writeup (15%)
(5%) - Introductory paragraph explaining what you did. Even if it is required work.
(10%) - Code is clear and well commented.