Reversing a .wav File

As long as there have been audio recordings, there have been secret messages (and unfounded rumours of secret messages) hidden in the audio. One common technique artists have used over the years is to add a backwards track to a recording. This can sound strange (and even a bit eery!). While some reversed lyrics can be a bit controversial, others are simply used for fun or because it sounds cool. But how can you know if a song has some scary hidden meaning or is simply playful? We don't have a lot of record players around we can play in reverse, but we can write a program to reverse the sound in a file!

Unfortunately, it isn't as simple as just reversing the bytes in a sound file to make the file play backwards. There are lots of different types of audio files; in this project we will be working with .wav files. These are very simple, uncompressed files, but they still have a lot of metadata and an interesting data format. For instance, the first 44 bytes of a wav file are the **header** - a bunch of metadata about the file. The header has a format like this:

Byte Number	Purpose
0-3	Must hold the bytes that correspond to 'RIFF'.
4-7	Number of bytes in the file, after subtracting 8.
8-11	File type. If this isn't 'WAVE' we won't process the file.
12-15	Format chunk. Must be 'fmt' or we won't process it.
16-19	Length of format data (32 bit integer).
20-21	Format type. Make sure this is the 16-bit integer value 1.
22-23	Number of channels. This is a 16-bit integer!
24-27	The sample rate. This is a 32-bit integer.
28-31	Will be the sample rate * bits per sample * channels / 8.
32-33	(Bits per sample * channels) / data type size.
34-35	Bits per sample
36-39	Data header. Must be 'data'.
40-43	Data section size.

What a bunch of gobbledygook. How are we supposed to use this?

It is actually pretty easy. A .wav file is a subtype of 'RIFF' file. We aren't going to get into what those are (you can look it up yourselves). And, we are going to simplify this assignment a bit by only concentrating on reversing a subset of .wav files (the point of this assignment is to understand data types and pointers, not to make a robust .wav reverser). For our purposes, we will not reverse any file that doesn't have the following criteria:

- 'RIFF' in bytes 0-3.
- The integer in bytes 4-7 is the file size - 8.
- The file type in bytes 8-11 is 'WAVE'.
- The file has 2 channels.

- The format type is the value 1.

If your file has any other values, just exit and print a message saying why!

Otherwise, we want to reverse the data. However, we cannot simply reverse the bytes in the data section. Sound is **sampled** (measured) by a microphone at regular intervals. For instance, many of your files will show a sampling rate of 44,100 in bytes 24-27. This means the microphone took 44,100 samples per second. However, there may be more than one channel. If the recording were in stereo for instance, there would be two sets of samples stored (one for the left mic and one for the right). Samples are stored together. We will only work with two channel samples to make this easier. The data section will look something like this:

```
| 02 01 04 03 | 06 05 08 07 |...
|(left)(right)|(left)(right)|... More samples
|-- Sample1 --|-- Sample2 --|...
```

Almost all of our systems today are what we call **little endian**. This means that if we had a multi-byte value (for instance here we are using 16-bit or two-byte values for each measurement), that the most significant portion is stored to the right. So the '02 01' values would be interpreted as decimal 258. We usually think of numbers as **big endian**, meaning the most significant portion of the number is stored to the left (i.e. '01 02'). On a big endian system, '02 01' would evaluate to decimal 513. This makes sense, as the '02' would then be considered more significant, resulting in a larger number.

To reverse the samples then, we need to make the last sample the first, the second-to-last the second, etc. But, we need to keep our endianess. The above samples would then be reveresed to

```
...              | 06 05 08 07 | 02 01 04 03 |
... More samples |(left)(right)|(left)(right)| End of file
...              |-- Sample2 --|-- Sample1 --|
```

Notice how we reversed the sample order but didn't reverse the channel or byte orderings? If we do this correctly, the .wav file will play in reverse, with the same stereo setup as the original (we won't hear the speakers swapped!).

Holy cow, how do I actually do this?

It isn't really too hard. As the *Hitchhikers Guide to the Galaxy* says on its cover, "Don't Panic". Our first order of business is to read the .wav file in. Files are just bytes, so let's simply read it in to a byte array. We think of a `char` as a character; to the computer it is simply a byte. Some have a visible representation and some don't. Really, to the computer it is just a number (or alternatively, just some switches on and some off). I have given you a file I/O library. You will need to look at it carefully, and read its documentation to learn how to use it.

Be sure you can read and load files and understand how the library works *before* you begin trying to reverse a .wav. Once you are comfortable with the library, write a main method that will

- take a file name to read from the command line
- take a file name to write to from the command line
- read a .wav and validate it meets our criteria
- reverse the data section as described above
- save a new .wav file with the reversed data

Be careful **not** to overwrite the original file. You can be *reasonably*, but not completely, sure that your program is working if you play the new .wav file and it is the original reversed. Ideally, you would be able to reverse a file, reverse the reveresed file, and end up with an identical original file (if on a Linux system you can use the `cmp` command to compare and the `hexdump -C` command to view the files).

Structure

I have given you the `file_lib.h` and `file_lib.c` pieces of code. You will create

- `wav.c` - a library for working with wave files. It will have the code for the functions declared in the `wav.h` header.
- `wav.h` - a header for the wave file library that includes
  - a struct for a wave file header called a `wav_header`.
  - a struct for a `wav_file` that has a pointer to the header, the file size, and a pointer to the data
  - a function that takes the first 44 bytes of a file's contents and creates a header from it
  - a function that loads a wave file by calling the `read_file` function from the file library, then takes the returned bytes and creates a new `wav_file`. It must then set the header, data size, and pointer to the data section. It will then return a pointer to the new `wav_file`.
  - a function that takes a `wav_file` and a file path, and prepares a byte array of the file. It then calls upon our `write_file` function to write the new audio file to disk.
- `main.c` - the core file that will use the libraries I gave you and the one you created. The code for reversing the data samples should be in this file.

Create any helper functions you wish. **DO NOT CHANGE THE STRUCTURE OF ANY FILES I HAVE GIVEN YOU**. If you feel like you need to change something, talk to me and I will help you understand the project better.

Can I Use AI Assistants on this project?

No. In future assignments you will get the chance to code with an assistant. Not this project.

Can I work with a partner?

Please do. But only groups of 2. However, teams may talk to one another to understand the project better. This means you can draw pictures or diagrams, debate about concepts, review documentation together, etc. but cannot share source or pseudocode.

Grading Rubric

- Proper documentation and commenting
- Makes proper use of my file I/O library (this includes checks for errors as described)
- `wav.h` is correct and well commented
- `wav.c` is correct
- Program compiles with no errors or warnings
- Works to perfectly reverse a wave file as described
- Prints out the following information when it runs:
  - the input file name
  - the output file name

    - the sample rate
    - the file size
    - the number of channels
- Handles errors gracefully

Each of these criteria will receive: 10 points for no issues, -1 point per issue, minimum value of 3 (more than 7 issues is counted as 7 issues). However, if you turn in no code or no *gradable* code, you can still receive a lower score. For instance, if you turned in code that does not compile **at all**, you would receive zero points. The same if you don't make a "good faith effort". For instance, turning in a "HelloWorld" style program would not automatically get 10 points because it compiles correctly.