Program description:

The program will let you choose a wave file on your device, open it up and then choose whatever frequencies you want and modify their weights. After that you can save the file or save as.

Features:

* Displays a graph of the fourier transform of the file that gets updated to reflect the changes you made to it
* Displays a graph of the sound waves that gets updated to reflect the changes you made to frequencies as you make them
* Click on the fourier transform graph to select a frequency, then it lets you choose how much you’d like to multiply by or set an absolute value. Also lets you choose a radius around that point that you’d like to affect. Frequencies in that area of the point you chose get multiplied, by the chosen factor at the point itself and interpolated up/down to 1 at the end of the area.

The program supports WAVs with bit-depths of 8, 16, 24, or 32, and sample rates of 8000-48000. The program only supports WAVE\_FORMAT\_PCM files. Internally, samples converted to double-precision floats. When saving, conversion is done back to integers with dithering.

In memory, the PCM will be saved as a sequence of arrays, since the total memory can get too big to store in one contiguous block. Since the max new file length is 3600 seconds and the max frequency is 48kHz, that makes the max samples we might store equal to 3600 \* 48000. This will take 3600\*48000\*8 bytes of memory. Maybe capping each array in the sequence at a fifth of that will be a good way to go.

I should be wary of overflows when doing arithmetic. Using saturated arithmetic may be wise.

Apparently 8-bit WAVs don’t use two’s complement while larger sizes do. I should be wary of this, maybe even stop supporting 8-bit.

If I so choose, I can decide to support stereo by converting it to mono (the algorithm is really simple, just add the two matching samples together), and give a warning when you open a stereo file that it will be turned to mono. If I can figure out a way to do this but then convert back to stereo when you save, that’d be even better.

Possible algorithm for stereo to mono and back: for each sample, calculate how much of the sum of both samples is in the first and second channels. Such as: the first channel is 0.8 of the sum, the second is 0.2 of the sum. This is basically channel-1-sample / sum and the same for the other channel. When you convert back to stereo, you split each sample into two based on these weights and there you have it. This could be generalized to support any number of channels.

Roadmap:

1. Plan GUI

2. Ability to open wave files and save/save as.

3. Temporary GUI for choosing a frequency and a multiplier and applying the modification without the fancy graphs

4. Draw graphs

5. Make frequency selectable from graph

Notes about RIFF and WAV:

* Chunks start with a four-character-code identifier, followed by a 4-byte size field, followed by the data
* The data starts at an address that’s aligned to divide by 2 relative to the start of the file
* If the chunk size is odd, there’s an extra padding byte at the end of it
* Chunk size does not include the size of the chunk header
* The file starts with a RIFF chunk. Afterwards, there’s a 4-byte character code which indicates what kind of RIFF (in our case, it should spell “WAVE”), and then you get a bunch of subchunks where all the real data is.
* Data can come in the form of one chunk with all the PCM, or a list of chunks that each contain a part of the PCM or define a length of silence. In the latter case, a cue chunk and a playlist chunk might appear to tell me the play order of these chunks, and even to tell me to loop some of them. I should support this. Internally I will store the PCM data separated into a sequence of arrays anyway, so I will just separate them when they originate from different chunks, and add metadata that says how much to loop this one. Silence chunks may turn non-silent so I will have to pop an error when the file size is suddenly too much, but data about the play order or looping shouldn’t change.
* The fact chunk may be useful for figuring out if there’s too many samples in the file
* Some WAVE files may have a byte alignment such that for example you have a 24-bit depth but each sample has a fourth padding byte such that samples come every 4 bytes. I should be wary of this.
* Should nChannels be less than the number of bits set in dwChannelMask, then the extra (most significant) bits in dwChannelMask are ignored (in WAVEFORMATEXTENSIBLE). Might be relevant, might not be.
* WAVE\_FORMAT\_EXTENSIBLE is indeed a different format tag than WAVE\_FORMAT\_PCM, and it is required for sample rates above 16-bit from the sounds of it. I’ll support them both.
* It looks like format chunks used to be different. I should make sure I’m reading the up-to-date format and maybe even supporting the old one too depending on how much it’s still in use (it isn’t).