WAVE File Format

CS 245

Spring 2018

1 Digital representation of audio

An audio signal consists of a continuous stream of voltage values. To store the signal digitally, samples of the signal are made at regular intervals. The quality of the sampling is determined by two parameters: (1) the *sampling rate*, and (2) the *sample resolution.* The sampling rate is the number of samples made per second, and the sample resolution is the number of bits used to represent each sample. The rates and resolutions used depend on the situation: a high fidelity recording of an orchestra would use greater values than a recording of a human voice for use with a cell phone.

The WAVE file format is the most common way to store uncompressed (raw) audio data — although the format does allow for a limited number of compression schemes. Arbitrary sampling rates are supported by WAVE files, but only 8 and 16 bit sample resolutions are offically part of the format. In addition, mono and stereo (one and two channel) data is officially supported. The format is easily extended, albeit not officially, to include other sample resolutions (such as 24 bit integer resolution, and 32 bit floating point resolution) and more than two channels.

2 Format overview

A WAVE file is organized in a hierarchical structure of units, called *chunks.* Each chunk contains a four byte label, a four byte size value, followed by a data section of the specified size. The data section of a chunk may contain a list of other chunks. See figure 1.

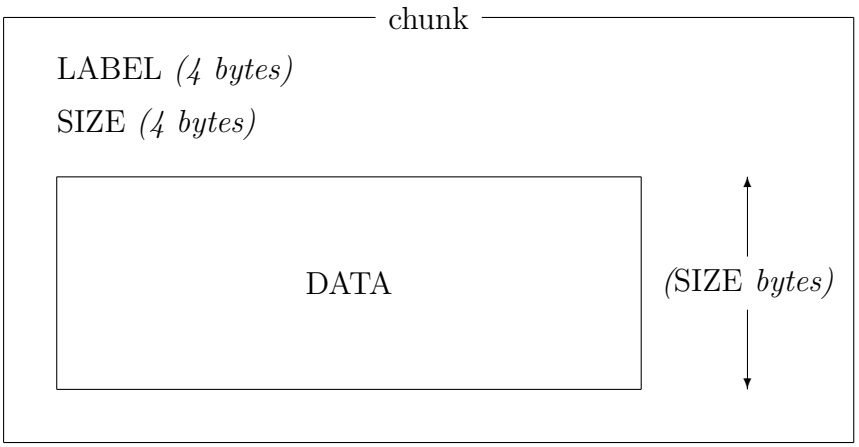

Figure 1: The fundamental unit of a RIFF file: the chunk.

Specifically, a WAVE file has an outermost chunk, labeled `"RIFF"` (a WAVE file is considered to be one particular type of a more general file type called a RIFF file). The data of the `"RIFF"` chunk consists of at least three items: (1) a 4 byte file type tag `"WAVE"`, (2) a `"fmt "` chunk (note the space), and (3) a `"data"` chunk. See figure 2. Other chunks may be present.

2.1 Format chunk

The data portion of the `"fmt "` chunk consists of a 16 byte data structure containing the sampling rate, resolution, and other pertinent information. In C++, the structure has the following form.

```
struct {
  uint16 audio_format;       // = 1
  uint16 channel_count;      // = 1 or 2
  uint32 sampling_rate;      // = 8000, 44100, etc.
  uint32 bytes_per_second;   // = (see below)
  uint16 bytes_per_sample;   // = (see below)
  uint16 bits_per_sample;    // = 8 or 16
};
```

Here `uint16` (`uint32`) is an unsigned 16 (32) bit integer in little endian format. On 32 bit and 64 bit Intel–based machines (which use little endian byte ordering), we may use

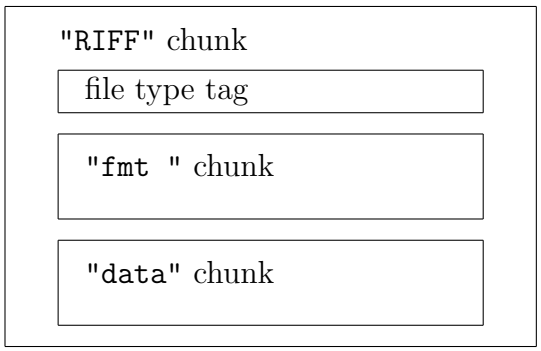

Figure 2: RIFF hierarchy of a WAVE file.

```
uint16 = unsigned short
uint32 = unsigned int
```

The `audio_format` field in the above structure indicates the format of the audio data; for us, this will be the value 1, which indicates uncompressed data. The `channel_count` field stores the number of channels (either 1 or 2). The `sample_rate` and `bits_per_sample` fields store the sampling rate and resolution of the data, respectively. The sampling resolution will have value of either 8 or 16. The two fields `bytes_per_second` and `bytes_per_sample` are derived from the sampling rate and resolution, and have the values

```
bytes_per_sample = channel_count*(bits_per_sample/8)
bytes_per_second = sampling_rate*bytes_per_sample
```

These fields are intended for use with compressed data; however, they give redundant information in the case of uncompressed data.

2.2 Data chunk

The data portion of the `"data"` chunk of a WAVE file stores the actual audio data. The format in which the data is stored depends on (1) the sample resolution, and (2) the number of channels.

Note that the size value of the `"data"` chunk is the number of *bytes* used by the audio data. Also, the size value for the outermost `"RIFF"` chunk will usually have a value of 36 plus the number of bytes used by the audio data.

2.2.1 8 bit mono data

For 8 bit mono samples, the data consists simply of an array of one byte samples, starting with the first sample

sample 0 *(1 byte)*	sample 1 *(1 byte)*	...

Each sample is stored as an *unsigned* 8 bit integer; in C or C++, this would be the type `unsigned char`. The midpoint of a sample value (0 voltage) is the value of 128. Thus to convert a sample s to a (signed) signal value V, we apply the formula

$$V = s - 128$$

The signal value is in the range $[-128 \ldots 127]$.

2.2.2 16 bit mono samples

The data in this case is also stored as a simple linear array of samples. However, the each sample is stored as a *signed* 16 bit integer in little–endian format.

sample 0 *(2 bytes)*	sample 1 *(2 bytes)*	...

In C/C++ on an Intel–based machine, each sample has type `short`. Unlike the 8 bit sample format, the midpoint is 0, so no conversion needs to be done to obtain the signal value from a sample value.

2.2.3 8 bit stereo samples

Stereo samples have two channels: one for the left channel, and one for the right. The data is stored in channel–interleaved format. That is, the left channel of the first sample is followed by the right channel of the first sample, which in turn is followed by the left channel of the second sample, which is followed by the right channel of the second sample, et cetera. To avoid confusion between a sample consisting of a left/right pair, and an individual channel sample, a left/right pair called a *frame*.

frame 0		frame 1		...
left *(1 byte)*	right *(1 byte)*	left *(1 byte1)*	right *(1 byte)*	...

Each channel of a frame is stored as an 8 bit unsigned integer, as described previously.

2.2.4 16 bit stereo samples

Just as with the 8 bit stereo samples, the data is organized in channel–interleaved format, but with the individual values being signed 16 bit integers.

frame 0		frame 1		...
left *(2 bytes)*	right *(2 bytes)*	left *(2 bytes)*	right *(2 bytes)*	...

3 Practical considerations

Although there is a hierarchical structure to a WAVE file, it is more convenient to view the structure as a header followed by the audio data. In this point of view, and with Intel architecture, the header of the file consists of the following C++ structure.

```
struct {
  char           riff_label[4];    // (00) = {'R','I','F','F'}
  unsigned       riff_size;        // (04) = 36 + data_size
  char           file_tag[4];      // (08) = {'W','A','V','E'}
  char           fmt_label[4];     // (12) = {'f','m','t',' '}
  unsigned       fmt_size;         // (16) = 16
  unsigned short audio_format;     // (20) = 1
  unsigned short channel_count;    // (22) = 1 or 2
  unsigned sampling_rate;          // (24) = (anything)
  unsigned bytes_per_second;       // (28) = (see above)
  unsigned short bytes_per_sample; // (32) = (see above)
  unsigned short bits_per_sample;  // (34) = 8 or 16
  char           data_label[4];    // (36) = {'d','a','t','a'}
  unsigned       data_size;        // (40) = # bytes of data
};
```

The offset of each field within the structure is given in parentheses, and the total size of the structure is 44 bytes. The audio data (in the format indicated in the previous section) the follows immediately after the header.

For example, if we have a valid WAVE file, named (say) `"file.wav"`, that stores CD quality audio data (16 bit stereo), we may read it in with the following code.

```
  fstream in(file_name,ios_base::in|ios_base::binary);
```

```
char header[44];
in.read(header,44);
unsigned size = *reinterpret_cast<unsigned*>(header+40);
short *data = new short[size/2];
in.read(reinterpret_cast<char*>(data),size);
```

(this assumes that the header file `fstream` has been included).