Waveform Audio File Format (WAV)

8/16-Bit, Mono/Stereo PCM

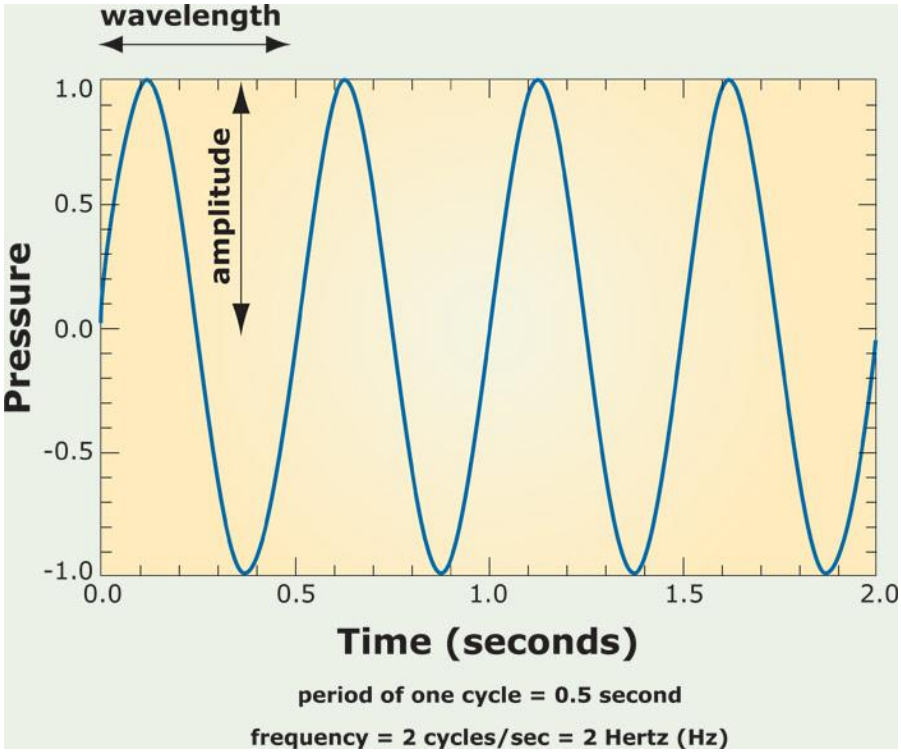
Abstract

In this paper, we will investigate the most common way uncompressed digital Pulse-code modulation (PCM) audio is stored on computer systems. Particularly, we will analyze and discuss how the WAV file format stores 8/16 bit Mono/Stereo Digital PCM signals. Example files are provided in **Visual Studio 2010** format.

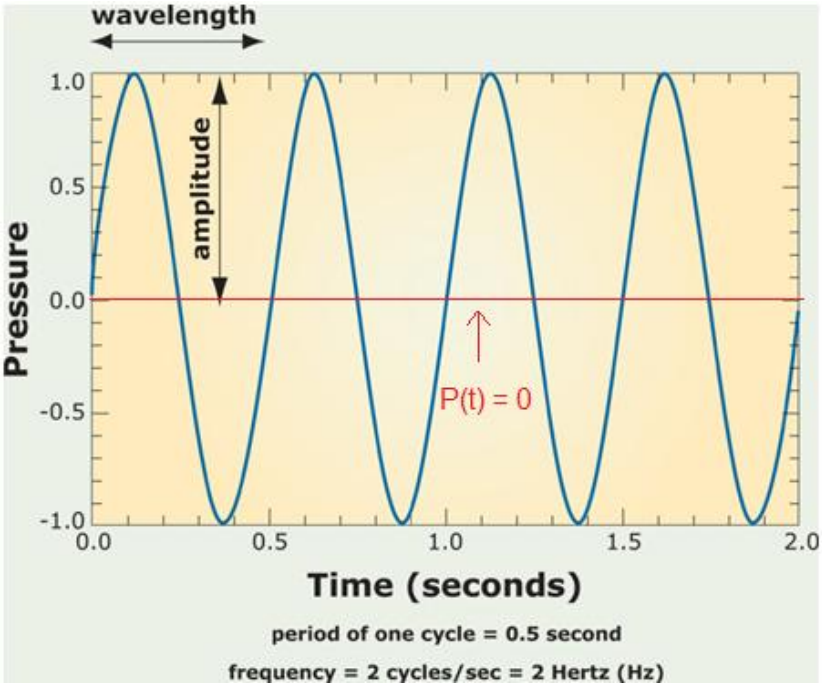
Introduction

The atmosphere contains many molecules. These molecules are too small for us to see. However, they produce the force we feel when there is a strong wind and the sound we hear when a person speaks. Sound occurs when the molecules in the atmosphere are re-ordered by an external force. This re-ordering process changes the pressure in the atmosphere and we hear sound as our ears re-adjust to the new pressure.

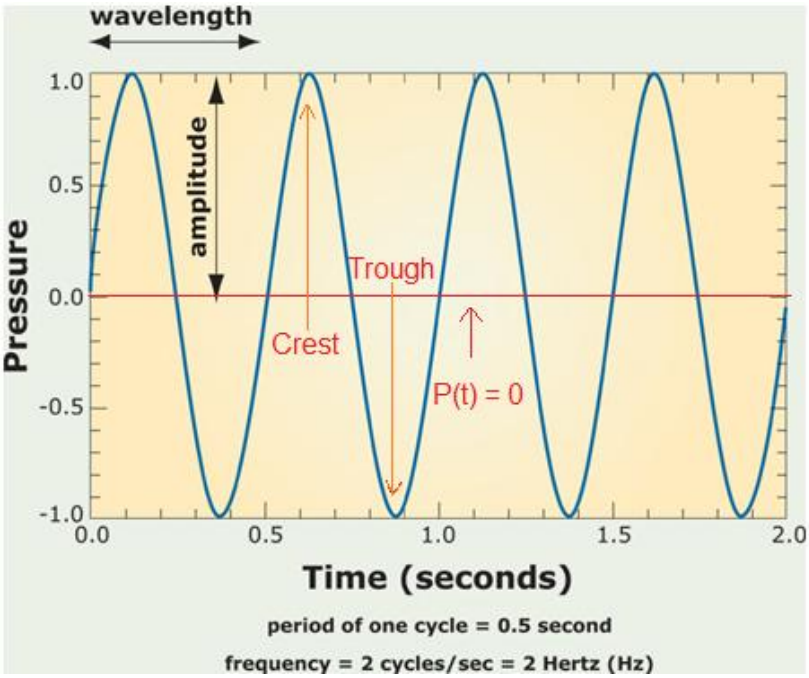
Electrical circuits, such as speakers, record the change in pressure anywhere from 8,000 to about 5,644,80 times per second. Audio CDs, for example, store digital audio signals that was recorded, or sampled, at about 44,100 times per second, or 44.1 kHz. To understand how audio pressure is sampled by electrical circuits, we observe a simple snapshot of sound wave:



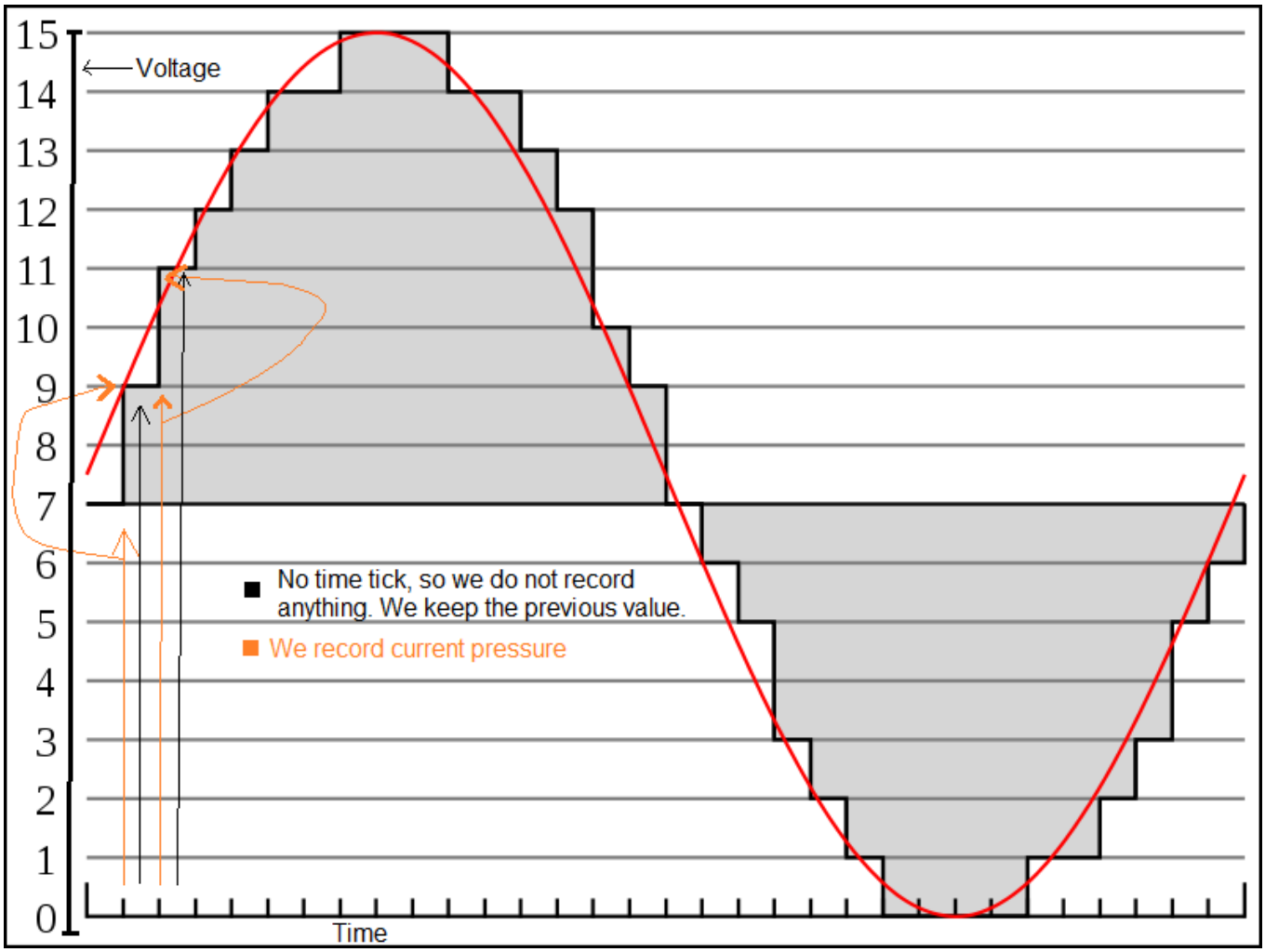
From the diagram above, we see that pressure is continuously changing over time. If we take the line p(t) = 0 (a horizontal line with the value p = 0 for every time) as our origin, then when our object begins to vibrate, or harmonically oscillate, it will move back and forth between this line:



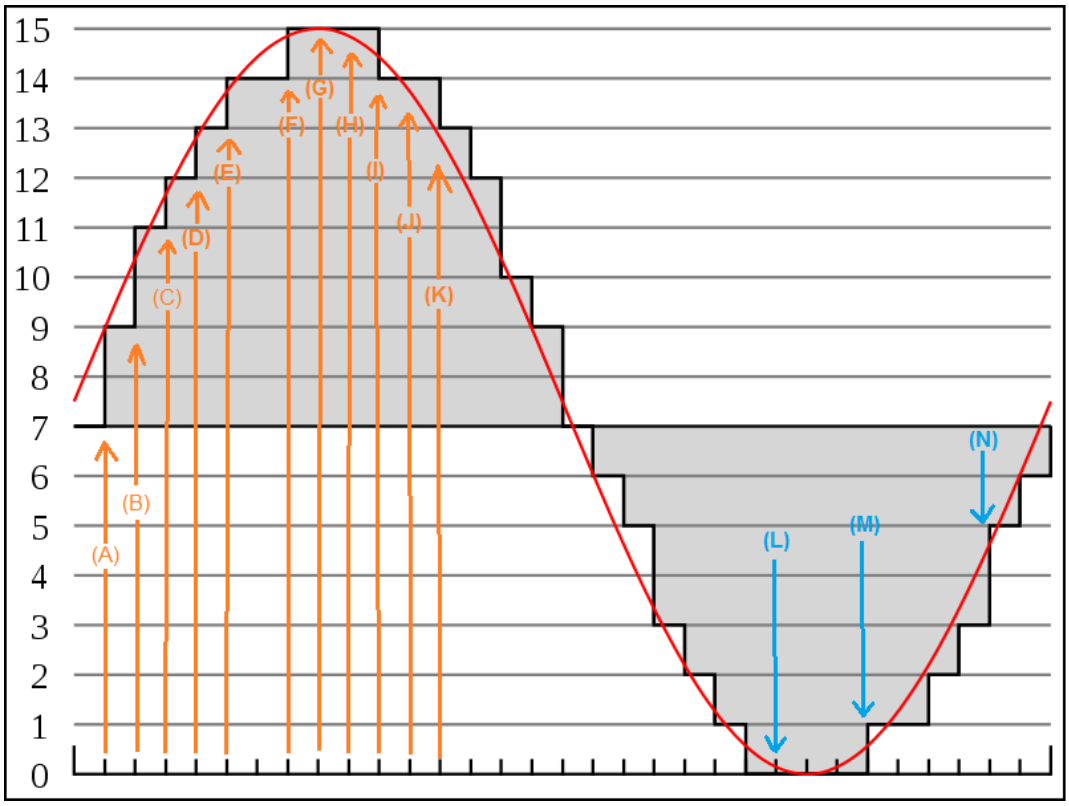
As the object vibrates back and forth, the pressure in the region it moves changes. If our object is vibrating from top to bottom, then we will notice the **crest** of our way at the top. However, when it moves towards the bottom, then we notice a **trough** (opposite of a crest) at the bottom. Finally, we determine how "loud" , or the intensity, of the sound we hear by the **amplitude** of the wave:



Electrical circuits, such as speakers, record these pressure changes about 44,100 times per second for audio CD/high quality sound. Because it is impossible to sample every single point of the continuous sound waves, we record the pressures every few ticks:



The recorded voltage is passed to an **Analog-to-Digital (AD)** converter, which then encodes a binary digital pattern to represent each voltage value. Typically, the values of the voltage can be encoded using 16-bits, which can represent 2^{16} = 65536 discrete voltage levels, or values. However, it is also possible to encode audio samples using 8-bits, 2^{8} = 256 discrete voltage levels, or values. The discrete binary patterns the Analog-to-Digital converter outputs are called **Pulse-code modulation (PCM)**, and it is the standard way to represent analog audio signals, or voltage levels.

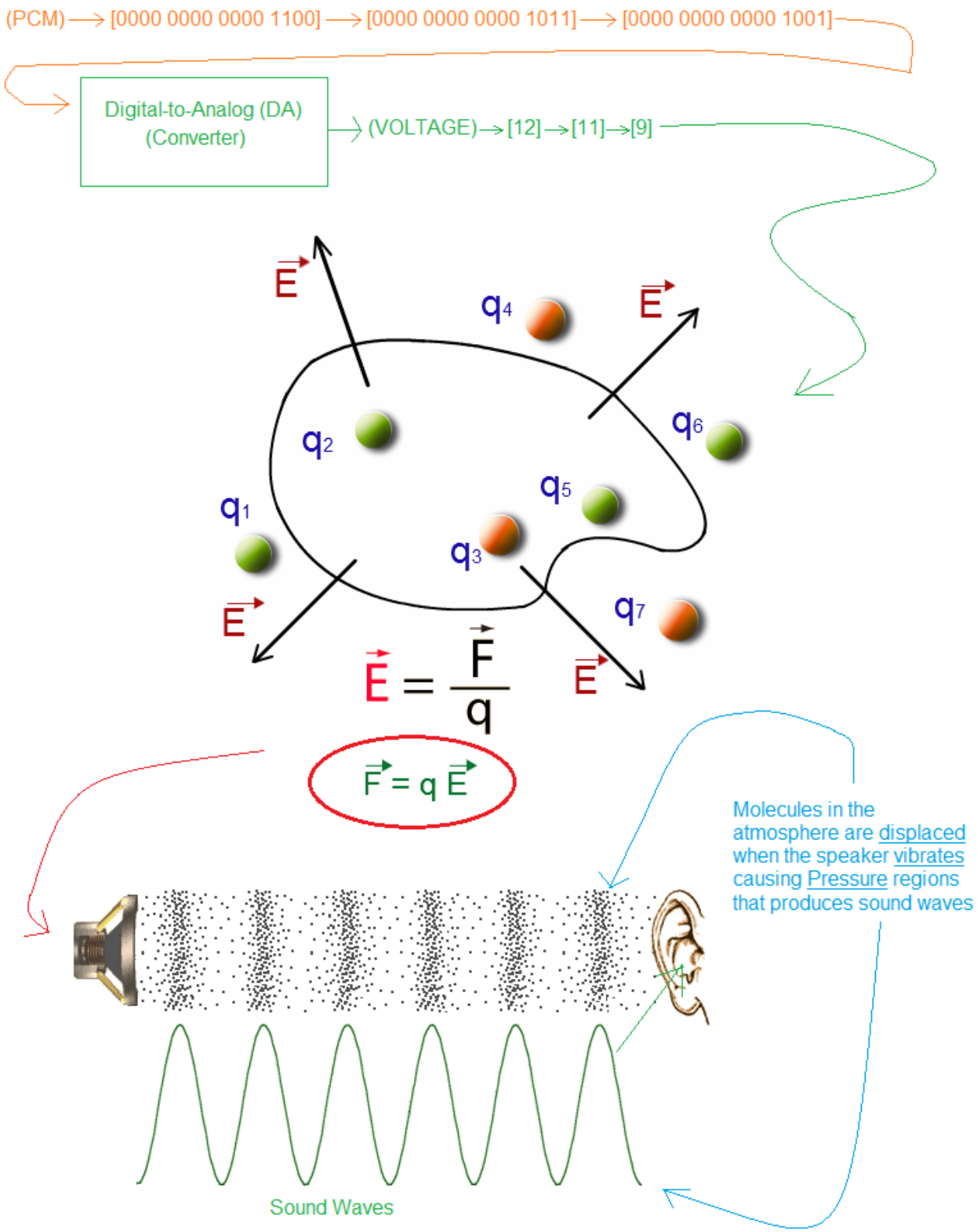


Pulse-code modulation (PCM)

(A) 0000 0000 0000 1001 = 9
(B) 0000 0000 0000 1011 = 11
(C) 0000 0000 0000 1100 = 12
(D) 0000 0000 0000 1101 = 13
(E) 0000 0000 0000 1110 = 14
(F) 0000 0000 0000 1111 = 15
(G) 0000 0000 0000 1111 = 15
(H) 0000 0000 0000 1111 = 15
(I) 0000 0000 0000 1110 = 14
(J) 0000 0000 0000 1110 = 14
(K) 0000 0000 0000 1101 = 13

(L) 0000 0000 0000 0000 = 0
(M) 0000 0000 0000 0001 = 1
(N) 0000 0000 0000 0101 = 5

The PCM codes, or binary pattern, generated by the analog-to-digital converter can be stored into a file on a computer for later playback. To reproduce the sound waves, the PCM codes, or binary values, will go through a **Digital-to-Analog (DA)** converter, at which point the digital values (PCM binary patterns) will be converted back to the equivalent voltage, or analog, value. The generate voltage will produce an electric field and the electric field will produce a force on the charged particles, which will in turn move and produce current. If the current is passing through a coiled wire, such as those in speakers, then the electric field will produce a perpendicular force on the coiled object (given by the right-hand rule) which will cause it to move, or vibrate. The vibration will then produce sound waves approximately the same as the original sound waves as far as we are concerned.


(PCM) → [0000 0000 0000 1100] → [0000 0000 0000 1011] → [0000 0000 0000 1001]
Digital-to-Analog (DA) (Converter)
(VOLTAGE) → [12] → [11] → [9]
E
q4
E
q6
q2
q5
q1
q3
E
q7
E
E = F / q
F = q E
Molecules in the atmosphere are displaced when the speaker vibrates causing Pressure regions that produces sound waves
Sound Waves

PCM values can be stored in a file on a computer with no additional information (raw PCM file), or with some additional information that will help us determine how fast we must play back the audio (sampling rate), how many channels of audio we have (in case we are outputting to a stereo speaker with multiple output channels), and how many bits we are using to encode each channel of PCM audio data. We will mention these items again. However, we will begin by discussing how we can store our PCM audio data into a file with no additional information (raw PCM).

As we mentioned earlier, the analog-to-digital converter may output 8 or 16-bit PCM values, each one corresponding to one of the analog voltage values necessary to make the speakers work. Let's say we have a 2V (2-volt) speaker that samples the pressure in the atmosphere at a rate of 6 samples/sec, or 5Hz. Let us also say that our speaker's voltage varies by 0.00784V intervals from 0V, it minimum voltage, up to 2V, its maximum voltage. Therefore, we would expect the following voltage values/signals:

0V, 0.00784V, 0.01568V, 0.02352V, 0.03136V, ... 2V

Mathematically, we can state this as:

$$V(x) = 0.00784 \cdot x \ \{x \in \mathrm{Z}^{+} \mid 0 \leq x \leq 255\}$$

Because the values are varying from 0V to 2V at 0.00784V intervals, we have 256 possible voltage signals/values:

(1)	0V
(2)	0.00784V
(3)	0.01568V
(4)	0.02352V
(5)	0.03136V
	.
	.
	.
(256)	2V

If we start counting from zero (0), then this list can be expressed as:

(0)	0V
(1)	0.00784V
(2)	0.01568V
(3)	0.02352V
(4)	0.03136V
	.
	.
	.
(255)	2V

If we have more samples, then the number of items in this list can become quite large! Because computers have limited disk space and memory, it is more convenient to count these items using base 16, or hexadecimal. In base 16, we represent each of these voltage levels as (starting from 0):

(00)	0V
(01)	0.00784V
(02)	0.01568V
(03)	0.02352V
(04)	0.03136V
	.
	.
	.
(FF)	2V

Most computer systems, particularly in the USA, tend to use ASCII encoding, uses hexadecimals to represent American decimals, alphabets, and special symbols, the voltage sequence we have above would need 1-3 bytes (0 to 255) to represent each decimal number in base 10. However, using hexadecimals directly to represent these voltage values allows us to store each value in 1 byte. So it is more convenient and economical to list these items, or represent each voltage level, using base 16!

Going back to our 2V speaker recording at 6 samples/sec (6Hz), let suppose we get the following voltage recordings/levels over 3 seconds:

[T = 1 sec] 0V 0.02352V, 0.01568V, 0.01568V, 0.03136V, 0.00784V
[T = 2 sec] 0.0392V, 0.0784V, 0.01568V, 0.03136V, 0.0392V
[T = 3 sec] 0.01568V, 0. 01568V, 0.01568V, 0.5096V, 1.8032V, 0V

Using hexadecimals, we can represent these voltage levels as:

[T = 1 sec] 00, 03, 02, 02, 04, 01
[T = 2 sec] 05, 0A, 02, 04, 04, 05
[T = 3 sec] 02, 02, 02, 41, E6, 00

Therefore, if we wanted to play back the sound that produced these signals, we could store the following sequence into a file:

00 03 02 02 04 01
05 0A 02 04 04 05
02 02 02 41 E6 00

Because we have separated each second of the audio (audio samples) into separate lines, we can always determine the rate we must send these signals back to the sound card or digital-to-analog converter. In this case, if we open the audio file with the hex sequence provided above, then we would send 6 hex values to the sound card per second. This would allow the sound card to deliver the original 6Hz sound wave encoded.

In the end, the hex values will be interpreted in a base 2 system by the computer. The computer prefers base 2 because it is made up of many transistors which act as switching allowing current to pass through

or not at all. Modern computers are made up of millions of these switches. These switches determine what actual the computer will take, or how it will behave. Therefore, the base 16 representation we just provided for the storage of the voltage signals will eventually be interpreted as a binary sequence, or base 2. Since or voltage levels vary from 0V to 2V over 0.00784V intervals, there are 256 voltage levels to encode, as we mentioned previous. In binary, this means we need to use 8-bits, or eight 1s and 0s pattern to represent the digital values. Therefore, or hex sequence:

00 03 02 02 04 01
05 0A 02 04 04 05
02 02 02 41 E6 00

is:

Pulse-code modulation (PCM), 8-bit, Mono-channel:
0000 0000 0000 0011 0000 0010 0000 0010 0000 0100 0000 0001
0000 0101 0000 1010 0000 0010 0000 0100 0000 0100 0000 0101
0000 0010 0000 0010 0000 0010 0100 0001 1110 0110 0000 0000

using 8-bits (eight 1s and 0s pattern) for each hexadecimal value. We color code the items above as visual aid. In a larger unit of measurement, we say that 8-bits is equal to 1 byte. The binary sequence we display above are precisely the PCM values mentioned earlier, except we are using 8-bits to represent our analog voltage signals in this case because our 2V speaker varies from 0V to 2V over 0.00784V intervals, which means it can produce only 256 different voltage signals. We use a hex value to represent each voltage signal for storage and the hex sequence is interpreted in terms of base-2, or binary by the computer for processing. The hex sequence which digitally represents our analog voltage signals will be referred to as **PCM** signals henceforth.

Storing the PCM signals has a sequence of hex values line-by-line for each second of audio is short and can be useful. However, it is not very informative, particularly if we would like to output to multiple channels, such as on stereo speakers, or if we would like to represent greater voltage intervals , or levels. For example, if you have a 12V stereo speaker in your car, you can have multiple output and input ports. Stereo speakers usually have more than one speaker. Each speaker vibrates based on the electrical signals (analog signals) provided to it. Each speaker may produce a different part of the audio or music you are listening to, therefore creating a higher quality experience. Most stereo speakers come in pairs, a **left** speaker and **right** speaker. We call the left speaker channel 0 and the right speaker channel 1. An example of a label you may see for a stereo speaker product is:

Each speaker can produce sound, therefore a multi-channel microphone, or electrical circuit which samples the sound wave pressures over time, would need a way to represent the signal from each channel. For example, let us suppose we had a multi channel speaker analogous to the signal channel 2V speaker we mentioned earlier. Let's say the left and right channels are each 2V and vary from 0V to 2V over 0.00784V intervals, similar to before but with two speakers now. A signal can be recorded for both as the following shows:

Let's say the recording was made at 6 samples/sec (6Hz) for each speaker, let suppose we get the following voltage recordings/levels over 3 seconds:

Voltage signals to Channel 0 (Left Speaker)

[T = 1 sec] 0V 0.02352V, 0.01568V, 0.01568V, 0.03136V, 0.00784V
[T = 2 sec] 0.0392V, 0.0784V, 0.01568V, 0.03136V, 0.0392V
[T = 3 sec] 0.01568V, 0. 01568V, 0.01568V, 0.5096V, 1.8032V, 0V

Voltage signals to Channel 1 (Right Speaker)

[T = 1 sec] 0V 0.02352V, 0.01568V, 0.01568V, 0.03136V, 0.00784V
[T = 2 sec] 0.0392V, 0.0784V, 0.01568V, 0.03136V, 0.0392V
[T = 3 sec] 0.01568V, 0. 01568V, 0.01568V, 0.5096V, 1.8032V, 0V

Using hexadecimals, we can represent these voltage levels as:

[T = 1 sec] 00 00, 03 03, 02 02, 02 02, 04 04, 01, 01
[T = 2 sec] 05 05, 0A 0A, 02 02, 04 04, 04 04, 05 05
[T = 3 sec] 02 02, 02 02, 02 02, 41 41, E6 E6, 00 00

The channels are color coded for visual aid. Now if we were to store this hex sequence, then it would be stored as:

00 00 03 03 02 02 02 02 04 04 01 01
05 05 0A 0A 02 02 04 04 04 04 05 05
02 02 02 02 02 02 41 41 E6 E6 00 00

Where there is two hex values per voltage signal. One for the left (channel 0) and one for the right (channel 1). We alternate each voltage sample by alternating the colors red and black for visual aid. When converted to base 2, or binary, this hex sequence becomes:

Pulse-code modulation (PCM), 8-bit, Stereo-channel (2):

0000 0000 0000 0000 0000 0011 0000 0011 0000 0010 0000 0010 0000 0010 0000 0010 0000 0100 0000 0100 0000 00010000 0001
0000 0101 0000 0101 0000 1010 0000 1010 0000 0010 0000 0010 0000 0100 0000 0100 0000 0100 0000 0100 0000 01010000 0101
0000 0010 0000 0010 0000 0010 0000 0010 0000 0010 0000 0010 0100 0001 0100 0001 1110 0110 1110 0110 0000 0000 0000 0000

You can quickly see why it is more convenient to represent the PCM codes as hex sequences! Going back to the hex sequence:

00 00 03 03 02 02 02 02 04 04 01 01
05 05 0A 0A 02 02 04 04 04 04 05 05
02 02 02 02 02 02 41 41 E6 E6 00 00

If we were to load such a file with no additional information, then following the same reasoning as we did before, we would read each line as 1 second of audio. Since there are 12 samples per line, this would lead us to think that we must send 12 samples, or hex values, to the speaker per second. This would cause our speakers to vibrate and produce sound waves at 12 samples/sec, which would imply the original audio was sampled at 12 samples/sec, or 12 Hz. However, this was not the case! We sampled at 6Hz for each channel (the left and right channels), therefore, we would want one of every hex pair to be sent to the left and right speakers so that each speaker plays its own sample at 6Hz rather than a single speaker playing the entire thing as 12Hz, which would produce a different sound. Therefore, we need a better way, or a more informative way, to store our hex sequence so that we can distinguish between multiple channels and different levels (different bits) of audio. A very common method to store PCM audio signals is **WAV**, also known as Waveform Audio Format.

Waveform Audio Format (WAV)

Although Wikipedia is not always reliable, we begin with a definition from it to define WAV in a general sense:

Waveform Audio File Format (WAVE, or more commonly known as WAV due to its filename extension), (also, but rarely, named, Audio for Windows) is a Microsoft and IBM audio file format standard for storing an audio bitstream on PCs. It is an application of the Resource Interchange File Format (RIFF) bitstream format method for storing data in "chunks", and thus is also close to the 8SVX and the AIFF format used on Amiga and Macintosh computers, respectively. It is the main format used on Windows systems for raw and typically uncompressed audio. The usual bitstream encoding is the linear pulse-code modulation (LPCM) format.

That is a great and general definition of what a wav file is. However, in a more technical and less wordy manner, I will simply say:

A WAV file is a data structure proposed by Microsoft and IBM to store linear PCM audio signals for efficient playback and cross-platform sharing. The data structure of a wav file is defined by the RIFF specifications, which extends the IFF standards.

To describe the wav file format and data structure, we will began from the root standard, **IFF,** Interchange File Format.

Interchange File Format (IFF)

IFF was originally released in 1985. According to digitalpreservation.gov:

The format was brought to completion in 1985 by the electronic gaming company Electronic Arts; some of the information about IFF refers to the Commodore-Amiga computer company and three Commodore-Amiga staff members are listed as co-authors of a key piece of documentation. An outline of the history of Electronic Arts is provided by Wikipedia.

The IFF format was influential and is cited as the source of ideas and approach by the creators of other tagged formats, e.g., RIFF and SMF.

A preserved specification document of the IFF format can be found at http://www.martinreddy.net/gfx/2d/IFF.txt and the most important element from this document is embedded in the following describe, which was pulled from the document:

***Chunks** are the building blocks in the IFF structure. The form expressed as a C typedef is:*

```
typedef struct {
        ID        ckID;
        LONG      ckSize;  /* sizeof(ckData) */
        UBYTE     ckData[/* ckSize */];
        } Chunk;
```

We can diagram an example chunk a "CMAP" chunk containing 12 data bytes like this:

```
                  ----------------
        ckID:     |    'CMAP'    |
        ckSize:   |      12      |
        ckData:   | 0, 0, 0, 32  |   --------
                  | 0, 0, 64, 0  |    12 bytes
                  | 0, 0, 64, 0  |   ---------
                  ----------------
```

The fixed header part means "Here's a type ckID chunk with ckSize bytes of data."

The ckID identifies the format and purpose of the chunk. As a rule, a program must recognize ckID to interpret ckData. It should skip over all unrecognized chunks. The ckID also serves as a format version number as long as we pick new IDs to identify new formats of ckData (see above).

The following ckIDs are universally reserved to identify chunks with particular IFF meanings: "LIST", "FORM", "PROP", "CAT ", and " ". The special ID " " (4 spaces) is a ckID for "filler" chunks, that is, chunks that fill space but have no meaningful contents. The IDs "LIS1" through "LIS9", "FOR1" through "FOR9", and "CAT1" through "CAT9" are reserved for future "version number" variations. All IFF-compatible software must account for these 23 chunk IDs. Appendix A has a list of predefined IDs.

Therefore, an IFF file contains a single **Universal Chunk** defined by one of the four IDs:

(1) LIST
(2) FORM
(3) PROP
(4) CAT_ (CAT + 1 blank space)

The **Universal Chunk** contains repeating chunks, which we will denote as **Subchunks** which take one of the following values:

(01) ____ (4 blank spaces)
(02) LIS1
.
.
.
(10) LIS9
(11) FOR1
.
.
.
(19) FOR9
(20) CAT1
.
.
.
(28) CAT9

The data structure for these chunks are defined below:

Universal Chunk ID (4-bytes worth of ANSCII) ***Description:*** *This field is always 4 characters long. Spaces are appended if the word is not long enough: The possible values for this field are:* (01) LIST (02) FORM (03) PROP (04) CAT_ (CAT + 1 blank space)
Chunk Size (32-bit integer, 4-bytes) ***Description:*** *This field states the size of the data field (the next row below) as a 32-bit integer value.*
Subtype ID (4-bytes worth of ANSCII) ***Description:*** *This field states type of data will be stored in the Universal Chunk (the Chunk ID this field is a sub of).*
<Subchunk> (Sequence of Subchunks) ***Description:*** *This field contains a sequence of Sub chunks. See the next table for more details.*

Subchunk ID (4-bytes worth of ANSCII) ***Description:*** *This field is always 4 characters long. Spaces are appended if the word is not long enough: The possible values for this field are:* (01) ____ (4 blank spaces) (02) LIS1 . . . (10) LIS9 (11) FOR1 . . . (19) FOR9 (20) CAT1 . . . (28) CAT9
Subchunk Size (32-bit integer, 4-bytes) ***Description:*** *This field states the size of the data field (the next row below) as a 32-bit integer value.*
Subchunk Data ("Subchunk Size" (see above) number of bytes) ***Description:*** *This field contains the data of the "Subchunk". It's size is "Subchunk Size" bytes.*

We can demonstrate the specifications IFF defines through some examples.

Note

(1) We will be analyzing our example files with Hex Workshop 6.7. It is a great tool for manipulating binary data or to learn about the data structure of a file or binary application. If we seek further information about an executable program, such as how the data structures are being used (algorithms), then we can use a disassembler such as IDA Pro.

(2) We will present all hex sequences in these example hex dumps in **Big Endian** form.

Example IFF File #1, file containing a list of images:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4C	49	53	54	00	00	00	44	70	69	63	73	70	69	63	31	LIST...Dpicspic1
00000010	00	00	00	0C	45	69	6E	73	74	65	69	6E	2E	62	6D	70	....Einstein.bmp
00000020	70	69	63	32	00	00	00	0F	4C	6F	61	64	69	6E	67	5F	pic2....Loading_
00000030	69	6D	67	2E	67	69	66	70	69	63	33	00	00	00	0D	6D	img.gifpic3....m
00000040	79	5F	67	72	61	70	68	2E	6A	70	65	67					y_graph.jpeg_

The **Hex Dump** of example **File #1** is displayed above. We will analyze it step-by-step and see how it formulates the specifications defined by IFF. According to the IFF specification document:

Rules for Writer Programs

Here are some notes on building programs that write IFF files, which is much easier than reading them. If you use the standard IFF writer module "IFFW.C" (see "Support Software" in Appendix A), many of these rules and details will automatically be enforced. See the example program "Raw2ILBM.C".

% An IFF file is a single FORM, LIST, or CAT chunk.

% Any IFF-85 file must start with the 4 characters "FORM", "LIST", or "CAT ", followed by a LONG ckSize. There should be no data after the chunk end.

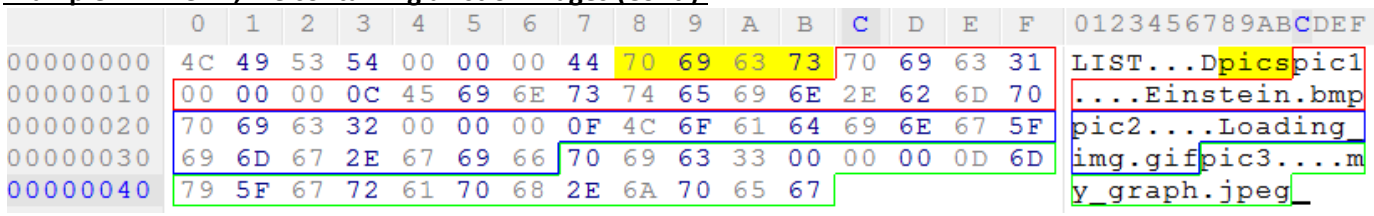
% Chunk types LIST, FORM, PROP, and CAT are generic. They always contain a subtype ID followed by chunks. These three IDs are universally reserved, as are "LIS1" through "LIS9", "FOR1" through "FOR9", "CAT1" through "CAT9", and " ".

% You must adhere to the syntax descriptions in Appendex A. E.g. PROPs may only appear inside LISTs.

Looking at our dump, we notice that our file obeys rule #1 because it begins with a **LIST** (4C 49 53 54) chunk (highlighted in yellow). Following the list chunk id, we have a 4-byte (long) integer type specifying the length of the chunk (00 00 00 44).We have highlighted 00000044 (68 decimal) worth of data in green. If you contain each hex sequence as "1-byte", then you will notice that there are 68 bytes, or (44 bytes in base 16). Therefore, there is no additional data after our LIST chunk. Our LIST chunk specified a chunk size of 0x44 (68 in decimal) and there are exactly 0x44 bytes after it. No additional data.). Therefore, our example file satisfies rule #2.

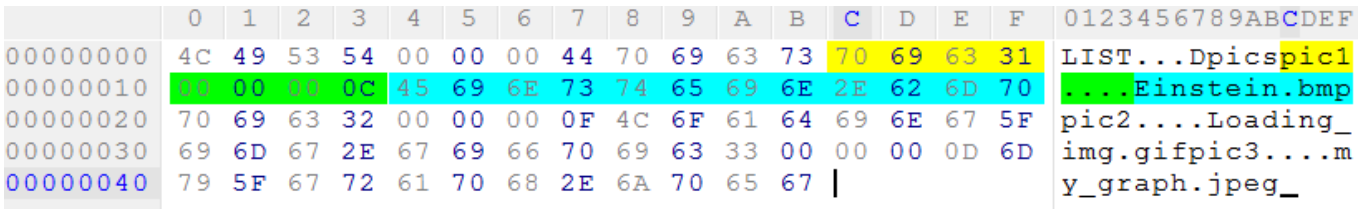
We continue our analysis:

Example IFF File #1, file containing a list of images (Cont.):



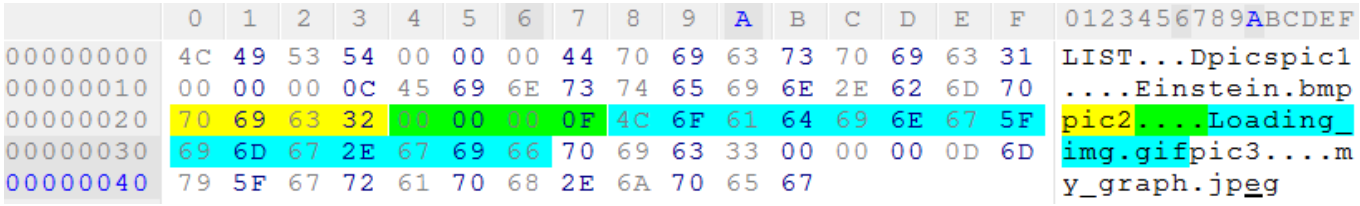
Following the chunk size value we just mentioned is a **Subtype** chunk with id **pics**. Therefore, we expect our example our IFF file to contain data relating to "pics". By naming our subtype, we can determine what kind of data the chunk contains and how we should parse it in our program. Also notice that the subtype id is 4-bytes as it should be). Therefore, our example file satisfies rule #3.

Next, we highlight our **Subchunks** in red, blue, and green. Looking over the first sub chunk:



We notice our first sub chunk contains the **Subchunk id** pic1 (4-bytes as expected), a **Subchunk size** of 00 00 00 0C (12 in decimal), which is also 4-bytes as defined by the standard, and the **Subchunk data** Einstein.bmp, which is precisely 0x0C bytes in length (12 bytes in decimal. each character is 1 byte).

Our second sub chunk shows:



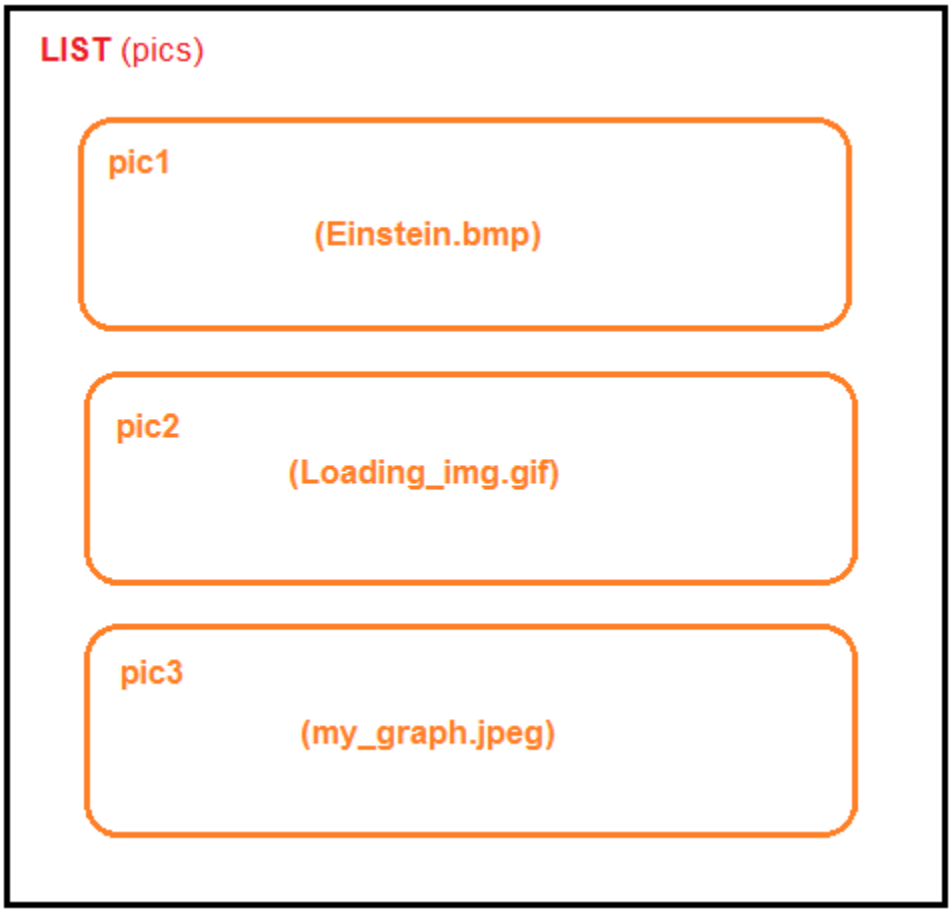
We notice our second sub chunk contains the **Subchunk id** pic2 (4-bytes as expected), a **Subchunk size** of 00 00 00 0F (15 in decimal), which is also 4-bytes as defined by the standard, and the **Subchunk data** Loading_img.gif, which is precisely 0x0F bytes in length (15 bytes in decimal. each character is 1 byte).

Our third and final sub chunk shows:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4C	49	53	54	00	00	00	44	70	69	63	73	70	69	63	31	LIST...Dpicspic1
00000010	00	00	00	0C	45	69	6E	73	74	65	69	6E	2E	62	6D	70	....Einstein.bmp
00000020	70	69	63	32	00	00	00	0F	4C	6F	61	64	69	6E	67	5F	pic2....Loading_
00000030	69	6D	67	2E	67	69	66	70	69	63	33	00	00	00	0D	6D	img.gifpic3....m
00000040	79	5F	67	72	61	70	68	2E	6A	70	65	67					y_graph.jpeg_

We notice our third sub chunk contains the **Subchunk id** pic3 (4-bytes as expected), a **Subchunk size** of 00 00 00 0D (13 in decimal), which is also 4-bytes as defined by the standard, and the **Subchunk data** my_graph.jpeg, which is precisely 0x0D bytes in length (13 bytes in decimal. each character is 1 byte).

Visually, our file data looks like:

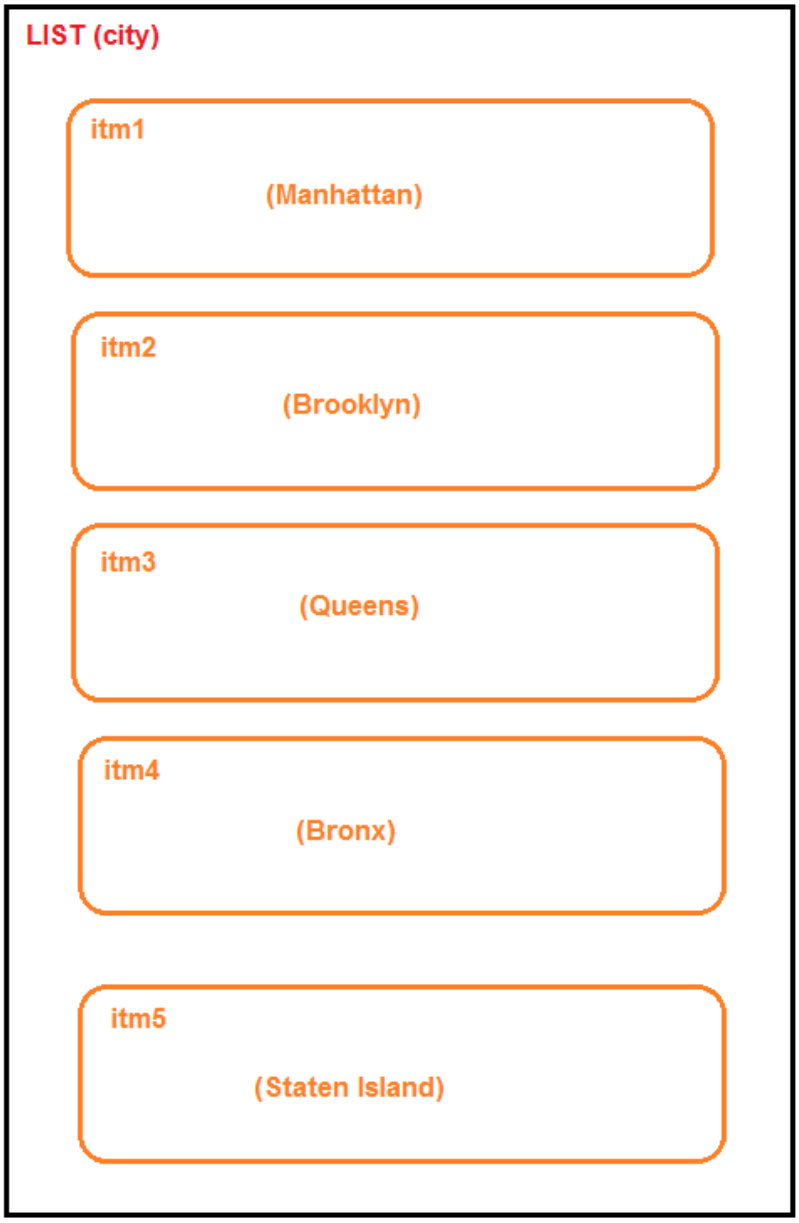


Let's take a second example:

Example IFF File #2, file containing a list of cities (Cont.):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4C	49	53	54	00	00	00	55	63	69	74	79	69	74	6D	31	LIST...Ucityitm1
00000010	00	00	00	09	4D	61	6E	68	61	74	74	61	6E	69	74	6D	....Manhattanitm
00000020	32	00	00	00	08	42	72	6F	6F	6B	6C	79	6E	69	74	6D	2....Brooklynitm
00000030	33	00	00	00	06	51	75	65	65	6E	73	69	74	6D	34	00	3....Queensitm4.
00000040	00	00	05	42	72	6F	6E	78	69	74	6D	35	00	00	00	0D	...Bronxitm5....
00000050	53	74	61	74	65	6E	20	49	73	6C	61	6E	64				Staten Island

Following the same procedure as before, we confirm that this file structure obeys the IFF standards and we end up with the following data structure:



The Interchange File Format was released in 1985. However, it served as the foundation for many other file format specification, particularly the one WAV is a subset of, **RIFF,** Resource Interchange File Format. The RIFF file format was released in August of 1991. Thought RIFF is newer than IFF, the foundations of RIFF are set through IFF's chunking standard.

Resource Interchange File Format (RIFF)

According to digitalpreservation.gov:

Multimedia Programming Interface and Data Specifications 1.0. IBM Corporation and Microsoft Corporation, August 1991. Available online, e.g., at http://www.tactilemedia.com/info/MCI_Control_Info.html

Multimedia Data Standards Update April 15, 1994 at http://www-mmsp.ece.mcgill.ca//Documents/AudioFormats/WAVE/Docs/RIFFNEW.pdf

The last major update to the RIFF standard was on September 16, 2004. The RIFF specification established the foundations for many of the multimedia content we see today. Some example of file structures that derive from RIFF (subsets of RIFF) are Bitmaps (BMP), Rich Text Format (RTF), **Waveform Audio File Format (WAVE)**, MCI Command Strings, and The Musical Instrument Digital Interface (MIDI). The successor of RIFF is the Advanced Systems Format (ASF).

There is nothing too special about RIFF. It is basically an extension of IFF. The most notable differences between the two formats, as far as we are concern for this paper, is the specific details of sub chunks and the universal chunk ids. The specification document for RIFF can be found at http://www.kk.iij4u.or.jp/~kondo/wave/mpidata.txt.

According to the specification document:

Resource Interchange File Format

> The Resource Interchange File Format (RIFF), a tagged file structure, is a general specification upon which many file formats can be defined. The main advantage of RIFF is its extensibility; file formats based on RIFF can be future-proofed, as format changes can be ignored by existing applications.
>
> The RIFF file format is suitable for the following multimedia tasks:
>
> - Playing back multimedia data
> - Recording multimedia data
> - Exchanging multimedia data between applications and across platforms

Therefore, it serves the goal we seek! We would like to store our audio's digital PCM values in a file structure that will permit us to unambiguously load and play back the sound. We saw how IFF describes it's file structure/chunks, now let us see how RIFF defines itself. According to the specification document:
Chunks

The basic building block of a RIFF file is called a chunk. Using C syntax, a chunk can be defined as follows:RIFF chunk;defined in C syntax

```
typedef unsigned long DWORD;
typedef unsigned char BYTE;

typedef DWORD FOURCC;       // Four-character code
typedef FOURCC CKID;        // Four-character-code chunk identifier
typedef DWORD CKSIZE;       // 32-bit unsigned size value

typedef struct {          // Chunk structure
   CKID     ckID;         // Chunk type identifier
   CKSIZE     ckSize;              // Chunk size
field (size of ckData)
   BYTE     ckData[ckSize];         // Chunk data
} CK;
```

A FOURCC is represented as a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters (ASCII character value 32) as required, with no embedded blanks.DWORD, in RIFF chunkRIFF files;FOURCC code inBYTE, in RIFF chunkFOURCCdatatype;inRIFFchunk

For example, the four-character code FOO is stored as a sequence of four bytes: 'F', 'O', 'O', ' ' in ascending addresses. For quick comparisons, a four-character code may also be treated as a 32-bit number.

The three parts of the chunk are described in the following table:RIFF chunk;IDRIFF chunk;size

Part	Description
ckID	A four-character code that identifies the representation of the chunk data data. A program reading a RIFF file can skip over any chunk whose chunk ID it doesn't recognize; it simply skips the number of bytes specified by ckSize plus the pad

byte, if present.

ckSize A 32-bit unsigned value identifying the size of ckData. This size value does not include the size of the ckID or ckSize fields or the pad byte at the end of ckData.

ckData Binary data of fixed or variable size. The start of ckData is word-aligned with respect to the start of the RIFF file. If the chunk size is an odd number of bytes, a pad byte with value zero is written after ckData. Word aligning improves access speed (for chunks resident in memory) and maintains compatibility with EA IFF. The ckSize value does not include the pad byte.

We can represent a chunk with the following notation (in this example, the ckSize and pad byte are implicit):RIFF chunk;notation

<ckID> (<ckData>)

Two types of chunks, the LIST and RIFF chunks, may contain nested chunks, or subchunks. These special chunk types are discussed later in this document. All other chunk types store a single element of binary data in <ckData>.

We see that RIFF is not much difference. The first notable change is that every RIFF document must begin with the **RIFF Chunk**. The RIFF Chunk comes in two forms, RIFF and **RIFX.** If the document beings with a **RIFF Chunk**, then we expect the 4-byte **Chunk Size** fields to be stored in **Little Endian** format**.** However, if the document begins with **RIFX**, then we expect the 4-byte **Chunk Size** fields to be stored in **Big Endian** format. Another chunk id that is similar in data structure is the **LIST Chunk**. According to the document, only RIFF (or RIFX) and the LIST chunks may contain nested chunks. Also, further investigation of the document states that the RIFF and LIST chunks have a "format" field, which is equivalent to the "Subtype" field of IFF Universal Chunks. Therefore, the RIFF and LIST chunks are define exactly the same as the Universal Chunks in IFF, with the exception of wording. Because RIFF documents begins with a RIFF or RIFX chunk, we can express the data structure of the Universal Chunks (RIFF, RIFX, and LIST) as:

Universal Chunk ID (4-bytes worth of ANSCII) ***Description:*** *This field is always 4 characters long. Spaces are appended if the word is not long enough: The possible values for this field are:* (01) RIFF (Document can begins with this) (02) RIFX (Document can begins with this) (03) LIST (Document cannot begins with this)
Chunk Size (32-bit integer, 4-bytes) ***Description:*** *This field states the size of the data field (the next row below) as a 32-bit integer value.*
Format ID (4-bytes worth of ANSCII) ***Description:*** *This field states type of data will be stored in the Universal Chunk (the Chunk ID this field is a sub of).*
<Subchunk> (Sequence of Subchunks) ***Description:*** *This field contains a sequence of Sub chunks. See the next table for more details.*

The RIFF specification is basically the same as the IFF specification as far as the Universal Chunks are concerned, with the except that every document must begin with RIFF or RIFX. The Sub chunk definitions is where the two standards diverge. There are special sub chunks for every file format in the RIFF standard. However, the definitions we are interested in for this paper is the Waveform Audio File Format (WAVE) section.

A WAVE audio file is defined by RIFF as:

The WAVE form is defined as follows. Programs must expect (and ignore) any unknown chunks encountered, as with all RIFF forms. However, <fmt-ck> must always occur before <wave-data>, and both of these chunks are mandatory in a WAVE file.

```
<WAVE-form> Ý
    RIFF( 'WAVE'
        <fmt-ck>            // Format
        [<fact-ck>]             // Fact chunk
        [<cue-ck>]          // Cue points
        [<playlist-ck>]         // Playlist
```

```
        [<assoc-data-list>]          //
 Associated data list
        <wave-data>   )          // Wave data
```

The WAVE chunks are described in the following sections.

WAVE Format Chunk

The WAVE format chunk <fmt-ck> specifies the format of the <wave-data>. The <fmt-ck> is defined as follows:

```
 <fmt-ck> :   fmt( <common-fields> <format-specific-fields> )

 <common-fields> :
     struct
     {
       WORD wFormatTag;           // Format category
       WORD wChannels;            // Number of channels
       DWORD   dwSamplesPerSec;         // Sampling rate
       DWORD   dwAvgBytesPerSec;        // For buffer estimation
       WORD wBlockAlign;       // Data block size
     }
```

The fields in the <common-fields> chunk are as follows:

Field	Description
wFormatTag	A number indicating the WAVE format category of the file. The content of the <format-specific-fields> portion of the fmt chunk, and the interpretation of the waveform data, depend on this value. You must register any new WAVE format categories. See Registering Multimedia Formats in Chapter 1, Overview of Multimedia Specifications, for information on registering WAVE format categories. Wave Format Categories, following this section, lists the currently defined WAVE format categories.
wChannels	The number of channels represented in the waveform data, such as 1 for mono or 2 for stereo.
dwSamplesPerSe	The sampling rate (in samples per second) at which each channel should be played.
dwAvgBytesPerS	The average number of bytes per second at which the waveform data should be transferred. Playback software can

	estimate the buffer size using this value.
wBlockAlign	The block alignment (in bytes) of the waveform data. Playback software needs to process a multiple of wBlockAlign bytes of data at a time, so the value of wBlockAlign can be used for buffer alignment.

The <format-specific-fields> consists of zero or more bytes of parameters. Which parameters occur depends on the WAVE format category–see the following section for details. Playback software should be written to allow for (and ignore) any unknown <format-specific-fields> parameters that occur at the end of this field.Samples, setting bits per sample

WAVE Format Categories

The format category of a WAVE file is specified by the value of the wFormatTag field of the fmt chunk. The representation of data in <wave-data>, and the content of the <format-specific-fields> of the fmt chunk, depend on the format category.Waveform files;format categories

The currently defined open non-proprietary WAVE format categories are as follows:

wFormatTag Value	Format Category
WAVE_FORMAT_PCM (0x0001)	Microsoft Pulse Code Modulation (PCM) format

The following are the registered proprietary WAVE format categories:

wFormatTag Value	Format Category
IBM_FORMAT_MULAW (0x0101)	IBM mu-law format
IBM_FORMAT_ALAW (0x0102)	IBM a-law format
IBM_FORMAT_ADPCM (0x0103)	IBM AVC Adaptive Differential Pulse Code Modulation format

The following sections describe the Microsoft WAVE_FORMAT_PCM format.

Pulse Code Modulation (PCM) Format

If the wFormatTag field of the <fmt-ck> is set to WAVE_FORMAT_PCM, then the waveform data consists of samples represented in pulse code modulation (PCM) format. For PCM waveform data, the <format-specific-fields> is defined as follows:

```
<PCM-format-specific> :
    struct
    {
      WORD wBitsPerSample;     // Sample size
    }
```

The wBitsPerSample field specifies the number of bits of data used to represent each sample of each channel. If there are multiple channels, the sample size is the same for each channel.

I will make a note on BNF definitions:

NOTE on BNF definitions/convention

Elements on a new line do not represent 'alternatives'. If 'alternatives' is meant, then a '|' (OR) operator will separate the alterative fields. Therefore:

```
<item>: <ckID> (
              <item_A>
              <item_B>
            }
```

is the same as:

```
<item>: <ckID> ( item_A> <item_B> )
```

Therefore, a new line does not represent 'alternatives'. Separating our items in new lines represents concatenation with a single space between the items. To express 'alternatives', the following conventions are adopted:

```
<item>: <ckID> (
              <item_A> |
              <item_B>
             }
```

This can also be represented in a single line as:

<item>: <ckID> (<item_A> | <item_B>)

Remember from the first quoting we provided, the items between the opening '(' and closing ')' represent the chunk data. The 'chunk size' field is always implicitly implied with this notation, as stated by RIFF.

According to the definition provided in the RIFF document, a wave file is basically composed of a single universal chunk, the **RIFF** or **RIFX** chunk. The RIFF or RIFX chunk defines the **FomatType id** with the value **WAVE**. The RIFF chunk then contains two **Subchunks**, a **fmt** and **data** chunk. The **fmt** chunk is defined in BNF as:

```
<fmt-ck>: fmt (
            <common-fields>
            <format-specific-fields>
          )

  <common-fields>:
     struct
      {
        WORD    wFormatTag;       // Format category (format of audio)
        WORD    wChannels;        // Number of channels
        DWORD   dwSamplesPerSec;  // Sampling rate
        DWORD   dwAvgBytesPerSec; // For buffer estimation
        WORD    wBlockAlign;      // Data block size
      }

  *IMPORTANT NOTE I WILL ADD*
  ----------------------------------------------------------------
  WORD   = unsigned 2-bytes --> unsigned 16-bits;
  DWORD  = unsigned 4-bytes --> unsigned 32-bits;
  ----------------------------------------------------------------
```

The <format-specific-fields> depends on the wFormatTag we are using. According to the specificiation document:

> The format category of a WAVE file is specified by the value of the wFormatTag field of the fmt chunk. The representation of data in <wave-data>, and the content of the <format-specific-fields> of the fmt chunk, depend on the format category.Waveform files;format categories
>
> The currently defined open non-proprietary WAVE format categories are as follows:
>
> wFormatTag Value Format Category

WAVE_FORMAT_PCM (0x0001) Microsoft Pulse Code Modulation (PCM) format

The following are the registered proprietary WAVE format categories:

wFormatTag Value	Format Category
IBM_FORMAT_MULAW (0x0101)	IBM mu-law format
IBM_FORMAT_ALAW (0x0102)	IBM a-law format
IBM_FORMAT_ADPCM (0x0103)	IBM AVC Adaptive Differential Pulse Code Modulation format

In our case, we will only support WAVE_FORMAT_PCM, or PCM. Therefore, we will expect the 32-bit wFormatTag field to take on the value 0x00000001. Because 32-bit integers are usually stored in Big Endian, we expect the value 0x10000000 for PCM wave in the actual file.

Also, if our wFormatTag is PCM, then the <format-specific-fields> (see above) will have 1 additional field, 'Bits per sample', Therefore, we need to add 1 field to our <format-specific-fields> section. As the document states:

'The following sections describe the Microsoft
WAVE_FORMAT_PCM format.

Pulse Code Modulation (PCM) Format

If the wFormatTag field of the <fmt-ck> is set to
WAVE_FORMAT_PCM, then the waveform data consists of
samples represented in pulse code modulation (PCM)
format. For PCM waveform data, the <format-specific-
fields> is defined as follows:

```
<PCM-format-specific>:
    struct
    {
      WORD wBitsPerSample;     // Sample size
    }
```

The wBitsPerSample field specifies the number of bits
of data used to represent each sample of each channel.
If there are multiple channels, the sample size is the
same for each channel.'

Therefore, our format chunk definition expands to:

```
<fmt-ck>: fmt (
            <common-fields>
            <format-specific-fields>
          )
```

```
<common-fields>:
    struct
    {
      WORD    wFormatTag;       // Format category (format of audio)
      WORD    wChannels;        // Number of channels
      DWORD   dwSamplesPerSec;  // Sampling rate
      DWORD   dwAvgBytesPerSec; // For buffer estimation
      WORD    wBlockAlign;      // Data block size
    }

<format-specific-fields>:
    struct
    {
      WORD wBitsPerSample;      // Sample size
    }

*IMPORTANT NOTE I WILL ADD #2*
-------------------------------------------------------
WORD   = unsigned 2-bytes --> uint16_t;
DWORD  = unsigned 4-bytes --> uint32_t;
-------------------------------------------------------
```

Now that we have the **fmt** chunk defined, we move onto the second chunk the WAVE RIFF format defines, the 'data' chunk.

The 'data' chunk of a WAVE file is defined as:

```
-------------------------------------------------------------
  The <wave-data> contains the waveform data. It is
  defined as follows:

   <wave-data>:   { <data-ck> | <wave-list> }

   // NOTE: The specification replaces '<wave-list>' above
   // with '<data-list>'. However, I believe that is a typo,
   // so I am following what I assume is correct and was meant.

   <data-ck>:     data ( <wave-data> )

   <wave-list>:   LIST (
                    'wavl' { <data-ck> | <silence-ck> ... }
                  )

    <silence-ck>: slnt ( <dwSamples:DWORD> ) // Count of silent samples

   Note:  The slnt chunk represents silence, not
   necessarily a repeated zero volume or baseline sample.
   In 16-bit PCM data, if the last sample value played
```

> before the silence section is a 10000, then if data is
> still output to the D to A converter, it must maintain
> the 10000 value. If a zero value is used, a click may
> be heard at the start and end of the silence section.
> If play begins at a silence section, then a zero value
> might be used since no other information is available.
> A click might be created if the data following the
> silent section starts with a nonzero value.

According to the definition above, the 'wave data' section is a sequence of 'data-ck' or 'wave-list' chunks. As the specification stated earlier, the only chunks that contain other chunks are the 'RIFF' and 'LIST' chunks.

The <wave-list> chunk is a 'LIST' chunk whose data section contains a sequence of <data-ck> or <silence-ck>. The silence tells us that no sound is to be played for '<dwSamples:DWORD>' number of audio samples (or data samples). Therefore, if our data section is a <wave-list> chunk which contains alternating sequence of <data-ck> and <silence-ck>, then we will know that whenever we encounter a <silence-ck>, we must send 0 (zero) amplitude data to the speaker for the next <dwSamples:DWORD> sample counts. Alternatively, we can stop all processing for x number of seconds that corresponds to the processing/playback of <dwSamples:DWORD> number of sample counts. Therefore, <silence-ck> can be used to compress wave files because we would represent <dwSamples:DWORD> number of samples as a single 32-bit type, as opposed to <wBitsPerSample:WORD> (see above) number of bits. However, many decoders skip this optional <silence-ck>, therefore we will also skip it. Since we wont support <silence-ck>, there wont be a need to add support for <wave-list> chunks.

The <data-ck>, on the other hand, is not a 'LIST' chunk. Therefore, it's data section will not contain other chunks. It's data section will contain the data defined by the 'fmt' chunk. This is the only part of the <wave-data> definition we will support because it is the only chunk type supported by most wave codecs.

Depending on the <wFormatTag:WORD>, <wChannels:WORD>, and <wBitsPerSample:WORD>, values defined in the <fmt-ck>, we could have various layouts. <wBitsPerSample:WORD> determines how many bits we use per channel, and the <wChannels:WORD> determines how many channels do we have per audio sample.

Some examples of layouts we can have for PCM are:
NOTE: Base 10, or decimal, is being used for these examples

```
----------------------------------------
+ PCM_DATA_LAYOUT +
----------------------------------------

(8-bit Mono):
------------------------------------------
<wFormatTag:WORD>          =  1 (PCM)
<wBitsPerSample:WORD>      =  8
<wChannels:WORD>           =  1
```

Sample 1 Sample 2 Sample 3 Sample 4
Channel 0 Channel 0 Channel 0 Channel 0

Where each 'Channel 0' is 8-bits.

(8-bit Stereo):

<wFormatTag:WORD> = 1 (PCM)
<wBitsPerSample:WORD> = 8
<wChannels:WORD> = 2

Sample 1 Sample 2 Sample 3
Channel 0 Channel 1 Channel 0 Channel 1 Channel 0 Channel 1

Where each 'Channel 0' is 8-bits and each 'Channel 1' is 8-bits.

(16-bit Mono):

<wFormatTag:WORD> = 1 (PCM)
<wBitsPerSample:WORD> = 16
<wChannels:WORD> = 1

Sample 1 Sample 2 Sample 3
Channel 0 Channel 0 Channel 0 Channel 0 Channel 0 Channel 0

Where each 'Channel 0' is 8-bits, therefore, 'Channel 0' = 16-bits.

(16-bit Stereo):

<wFormatTag:WORD> = 1 (PCM)
<wBitsPerSample:WORD> = 16
<wChannels:WORD> = 2

Sample 1 Sample 2
Channel 0 Channel 0 Channel 1 Channel 1 Channel 0 Channel 0 Channel 1 Channel 1

Where each 'Channel 0' is 8-bits and each 'Channel 1' is 8-bits. Therefore, 'Channel 0' = 16-bits and 'Channel 1' = 16-bits.

Since we won't support for <wave-list>, the definition of what we will implement will change from:

Standard definition

<wave-data>: { <data-ck> | <wave-list> }

<data-ck>: data (<wave-data> | <wave-list>)

<wave-list>: LIST (
'wavl' { <data-ck> | <silence-ck> ... }
)

<silence-ck>: slnt (<dwSamples:DWORD>) // Count of silent samples
---

will change to our implementation definition:

Our Implementation definition

<wave-data>: { <data-ck> }

<data-ck>: data (PCM_DATA_LAYOUT)

The final data structure of our WAV file format will be:

RIFF (or RIFX) (4-bytes worth of ASCII)
RIFF Chunk Size (32-bit unsigned integer, 4-bytes) ***Description:*** *This field will specify how much data follows. The final value will be* ***filesize - 8***
RIFF Format ID (4-bytes worth of ASCII) ***Description:*** *This field will take on the value* ***WAVE***
fmt_ (fmt + 1 blank space) (WAVE Subchunk #1 - 4 bytes worth of ASCII)
fmt_ Chunk Size (32-bit unsigned integer, 4-bytes) ***Description:*** *This field will specify how much data follows. The final value will be* ***filesize - 20***
AudioFormat (2-bytes unsigned integer) ***Description:*** *This field will take on the value of 0x0001 (1) for PCM*

NumberOfChannels (2-bytes unsigned integer) ***Description:*** *This field will vary depending on the audio*
SampleRate (4-bytes unsigned integer) ***Description:*** *This field will take on the value of 8000, 44,100, etc depending on the sample/sec*
ByteRate (4-bytes unsigned integer) ***Description:*** *This field will take on the value* ***SampleRate * NumChannels * BitsPerSample/8***
BlockAlign (2-bytes unsigned integer) ***Description:*** *This field is used by programs to do fast processing. It will take on the value* ***NumChannels * BitsPerSample/8***
BitsPerSample (2-bytes unsigned integer) ***Description:*** *This field will vary depending on the audio. Typical values are 8, 16, 24, or 32.*
ExtraParamSize (2-bytes unsigned integer) ***Description:*** *This field will vary depending on the audio. This should only be present for non-PCM.*
ExtraParamsData ("ExtraParamSize" bytes) ***Description:*** *This field will vary depending on the audio. This should only be present for non-PCM and if ExtraParamSize is not 0.*
data (WAVE Subchunk #1 - 4 bytes worth of ASCII)
data Chunk Size (32-bit unsigned integer, 4-bytes) ***Description:*** *This field will specify how much data follows.*

Audio data ("data Chunk size" bytes) ***Description:*** *This field will vary depending on the audio. It will be put in the layout described earlier.*

Method/Implementation

We used the C++ programming language to decode and play a WAV file. The program allows the users to adjust the playback and sample rate of the decoded data. The program user can **pause**, **play**, **stop**, adjust the **sample rate** by pressing the keys **1 -> 0** on the keyboard. Press the **1** key will play the decoded WAV file at its original sample rate. Finally, the program user can press **Escape** on the keyboard to close the program.

The program comes in two parts,

(1) Console window
(2) Graphical user window

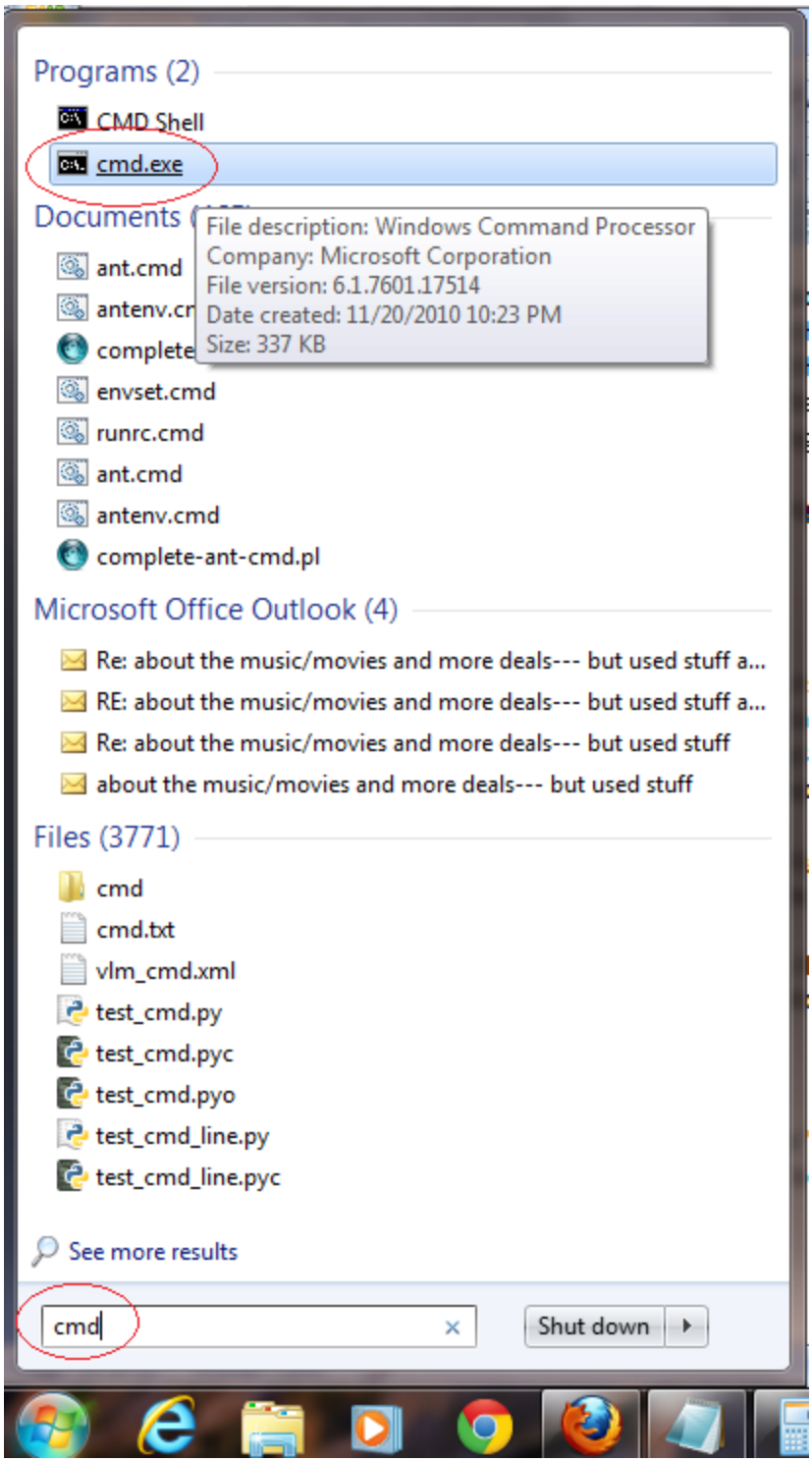
The console window displays additional information about the decoding process and WAV file information. When the users press a key or interact with the graphical window, the console window displays a line of text with the details of the action the user took. Finally, we the user take an action in the graphical window, a loading image is image is displayed while the settings are adjusted.

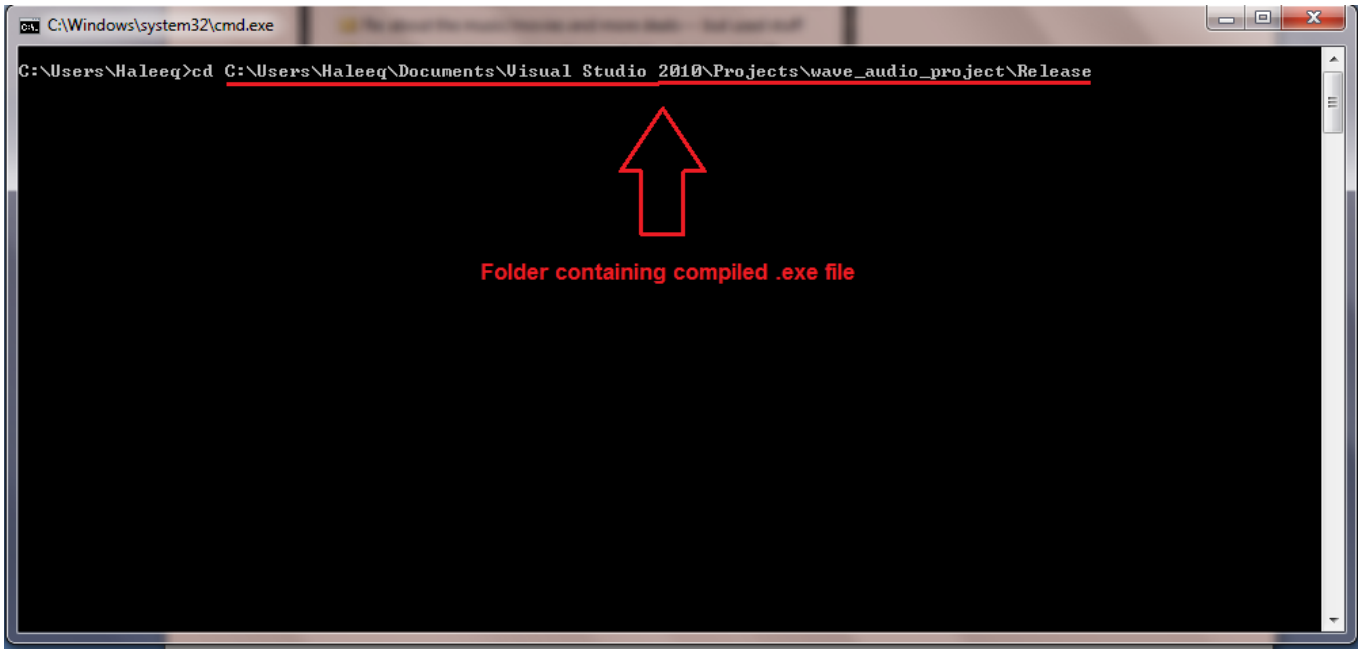
This program does not consider optimization or normalization of the algorithms/codes in any form. It is only meant to demonstrate the concept behind decoding and playing WAV audio files.

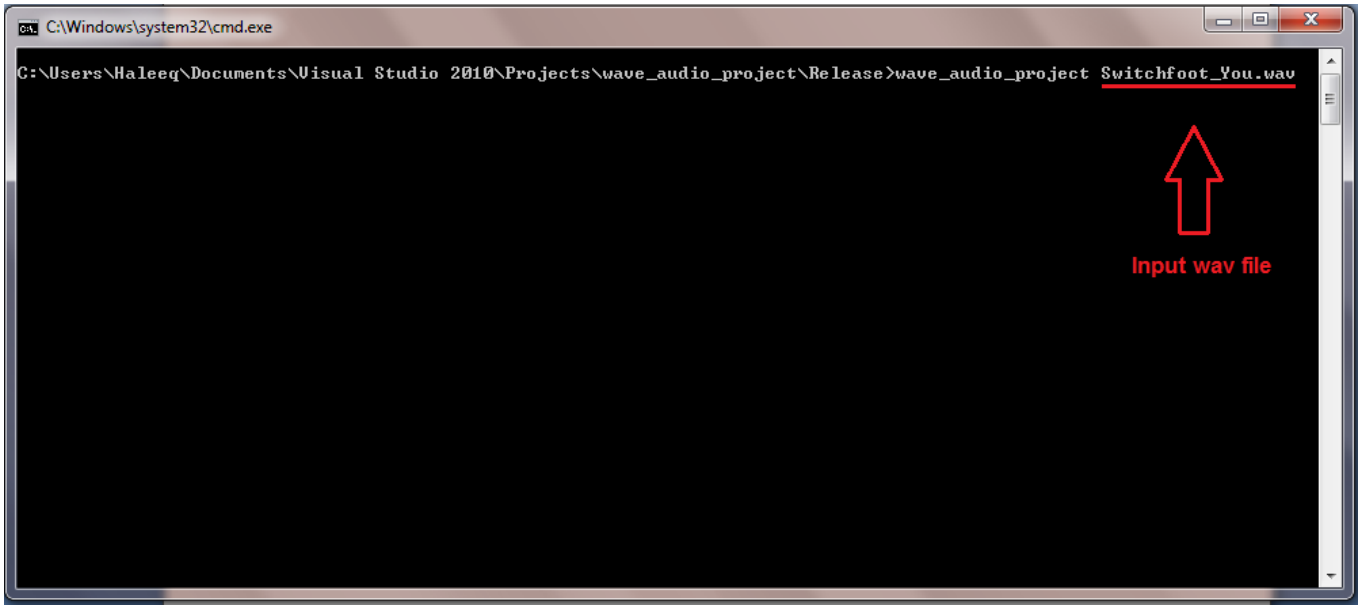
To run the program after compilation, the program user will need to run **command prompt** and enter the following command:

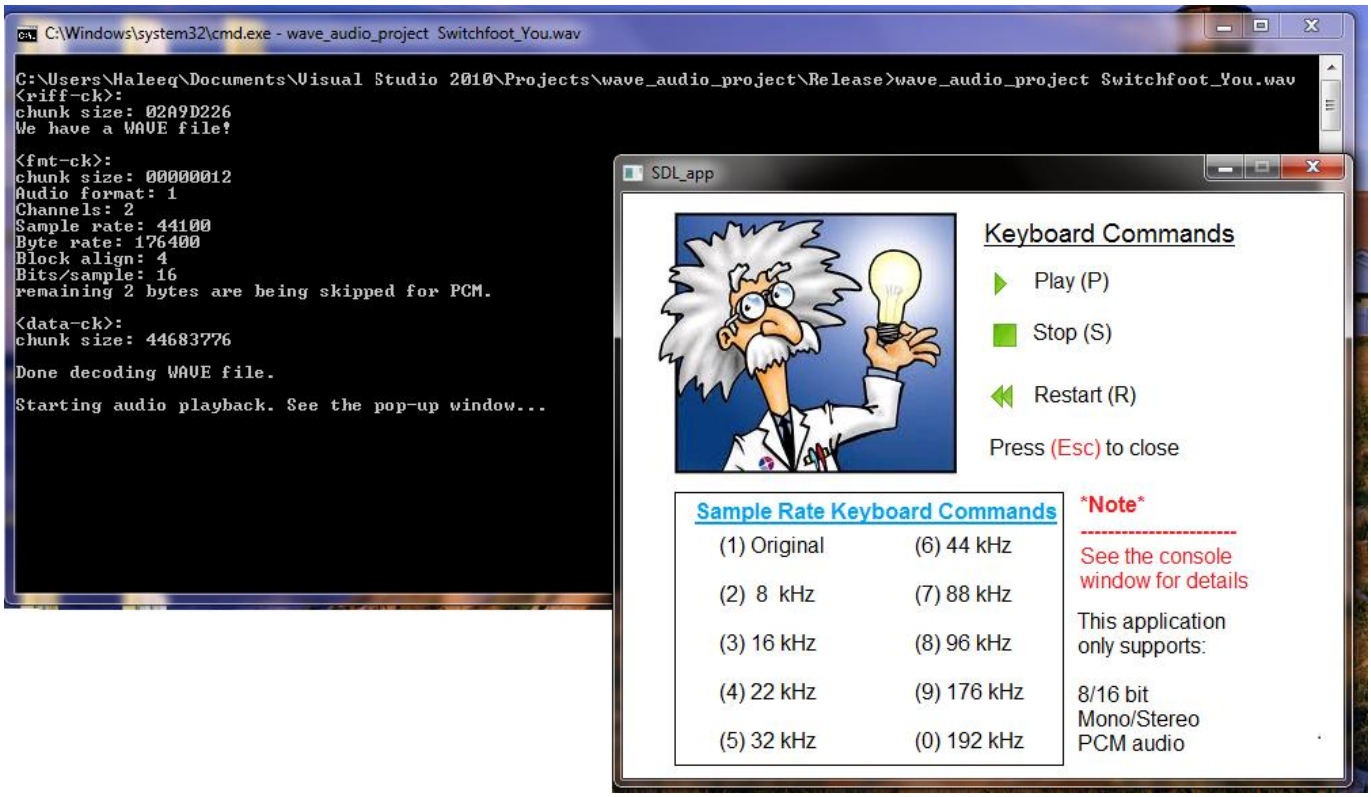
[command prompt]: **wave_audio_project** *INPUTFILE.WAV*

Where "wave_audio_project" is the name of the .exe file and **"**INPUTFILE.WAV" is the input wav audio file to load. An example of running the program is shown below:


Programs (2)
CMD Shell
cmd.exe
Documents
File description: Windows Command Processor
Company: Microsoft Corporation
File version: 6.1.7601.17514
Date created: 11/20/2010 10:23 PM
Size: 337 KB
ant.cmd
envset.cmd
runrc.cmd
ant.cmd
antenv.cmd
complete-ant-cmd.pl
Microsoft Office Outlook (4)
Re: about the music/movies and more deals--- but used stuff a...
RE: about the music/movies and more deals--- but used stuff a...
Re: about the music/movies and more deals--- but used stuff
about the music/movies and more deals--- but used stuff
Files (3771)
cmd
cmd.txt
vlm_cmd.xml
test_cmd.py
test_cmd.pyc
test_cmd.pyo
test_cmd_line.py
test_cmd_line.pyc
See more results
cmd
Shut down


C:\Windows\system32\cmd.exe
C:\Users\Haleeq>cd C:\Users\Haleeq\Documents\Visual Studio 2010\Projects\wave_audio_project\Release
Folder containing compiled .exe file


C:\Windows\system32\cmd.exe
C:\Users\Haleeq\Documents\Visual Studio 2010\Projects\wave_audio_project\Release>wave_audio_project Switchfoot_You.wav
Input wav file


C:\Windows\system32\cmd.exe - wave_audio_project Switchfoot_You.wav
C:\Users\Haleeq\Documents\Visual Studio 2010\Projects\wave_audio_project\Release>wave_audio_project Switchfoot_You.wav
<riff-ck>:
chunk size: 02A9D226
We have a WAVE file!
<fmt-ck>:
chunk size: 00000012
Audio format: 1
Channels: 2
Sample rate: 44100
Byte rate: 176400
Block align: 4
Bits/sample: 16
remaining 2 bytes are being skipped for PCM.
<data-ck>:
chunk size: 44683776
Done decoding WAVE file.
Starting audio playback. See the pop-up window...
SDL_app
Keyboard Commands
Play (P)
Stop (S)
Restart (R)
Press (Esc) to close
Sample Rate Keyboard Commands
(1) Original
(2) 8 kHz
(3) 16 kHz
(4) 22 kHz
(5) 32 kHz
(6) 44 kHz
(7) 88 kHz
(8) 96 kHz
(9) 176 kHz
(0) 192 kHz
Note
See the console window for details
This application only supports:
8/16 bit Mono/Stereo PCM audio


C:\Windows\system32\cmd.exe - wave_audio_project Switchfoot_You.wav
C:\Users\Haleeq\Documents\Visual Studio 2010\Projects\wave_audio_project\Release>wave_audio_project Switchfoot_You.wav
<riff-ck>:
chunk size: 02A9D226
We have a WAVE file!
<fmt-ck>:
chunk size: 00000012
Audio format: 1
Channels: 2
Sample rate: 44100
Byte rate: 176400
Block align: 4
Bits/sample: 16
remaining 2 bytes are being skipped for PCM.
<data-ck>:
chunk size: 44683776
Done decoding WAVE file.
Starting audio playback. See the pop-up window...
[Command - 7] Sample rate changed to 88200 Hz...
[Command - 7] Sample rate changed to 88200 Hz...
SDL_app
PLEASE WAIT
LOADing

The SDL and SDL Mixer libraries were used for the playback of the decoded audio data. All of the include and library files are provided in the zip file accompany this paper. However, fresh companies could be attained from:

(1) http://www.libsdl.org/download-1.2.php
(2) http://www.libsdl.org/projects/SDL_mixer

The files of interest are:

(1) SDL-devel-1.2.15-VC.zip (Visual C++)
(2) SDL_mixer-devel-1.2.12-VC.zip

Respectively.

Two additional external files were used:

(1) inttypes.h
(2) stdint.h

These files are also provided in the zip file. Detail comments are provided in the header file including these fields as to why they are being used. We use them to gain some additional C99 functionality defined in later C/C++ standards that the Visual C++ compiler does not support.

To Compile, open the **csc_211_audio_project.vcxproj** file with **Visual Studio 2010** or later and build the project. The include and library directories have been linked using the following scheme in the **project properties**:

(1) Additional Include Directory: $(ProjectDir)\include
(2) Additional Libraries Directory: $(ProjectDir)\lib\x86

Assuming they were not moved, then the compiler should link and compile just fine. To run the program, please use command prompt as instructed earlier.

Results

When the sample rate of the audio is changed, the audio becomes very high or low frequency. We notice that the audio begins to sound sort of like a "chipmunk", or as if one had inhaled some Helium and was trying to speak. This was very interesting and cool to play with. At just the right sample rates, you can get some pretty cool sound effects.

Conclusion

Alas, this experiment has been very interesting and incredibly informative. Before beginning this experiment, I had no idea how WAV files looked or how audio signals are digitally stored on computer systems. Though I have dealt with images before, I have never experimented or research into audio processing/storage. So this experiment was extremely informative. However, it also was tough! I attempted to gather some information on A and U-law PCM compression/data packing. However, I was unsuccessful in finding information on the data packing aspect of A and U-law PCM encoding. However, I did run into the G.711 and G.191 standard specification documents and managed to attain information and the algorithm to expand (decode) and compress A and U-law PCM audio.

I made several attempts to guess the data packing implied by the algorithm, however, I became very tired and decided further investigate this process sometime in the near future. I have left the expansion and compression algorithms for A and U-law that I attained from the G.191 standards. It is something definitely worth further investigating in the near future.

While working on the physical theory of wave mechanics, I became very interested in applying the same mathematics/method Albert Einstein did to derive his gravitation field equation. However, for sound in this case. Einstein's special and general relativity theories are of great interest to me. Though I decided to major in Chemistry when I graduated H.S. I have studied single variable Calculus, multi-variable/vector Calculus, Tensor Analysis, General Topology, and a little bit of Differential Geometry in my pursuit to understand the General Theory of Relativity. Though I failed many times over the course of three years to understand the theory after graduating H.S, I finally was able to understand a mathematical derivation of Einstein's gravitation field equations while studying from the book "Introduction to Tensor Calculus and Continuum Mechanics" by Dr. John H. Heinbockel of The University of Old Dominion, Virginia. He has the book available online for free at his website http://www.math.odu.edu/~jhh/counter2.html

After reading through Dr. Heinbockel's Tensor Calculus book, I moved on to the book "A first course in general relativity" By Dr. Bernard F. Schutz of the University of Max Planck Institute for Gravitational Physics, Potsdam-Golm. While working on the physical perspective of sound waves in the introduction, I was attempt to begin by instantaneously modeling sound waves as a scalar field when the air molecules are hypothetically not in motion and derive the associate vector field from the gradient of the scalar field when a an external force causes acceleration/motion in the atmosphere. I was attempted to see if it would be possible to model the atmosphere using fluid dynamics (since the atmosphere is composed of gas molecules after all) and derive an equation in terms of the stress-energy tensor to see where it leads me. However, I decided to put that off for another day.

Appendix A: ASCII Table (URL: http://www.ascii-code.com/)

ASCII control characters (character code 0-31)

The first 32 characters in the ASCII-table are unprintable control codes and are used to control peripherals such as printers.

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
0	000	00	00000000	NUL	&#000;		Null char
1	001	01	00000001	SOH	&#001;		Start of Heading
2	002	02	00000010	STX	&#002;		Start of Text
3	003	03	00000011	ETX	&#003;		End of Text
4	004	04	00000100	EOT	&#004;		End of Transmission
5	005	05	00000101	ENQ	&#005;		Enquiry
6	006	06	00000110	ACK	&#006;		Acknowledgment
7	007	07	00000111	BEL	&#007;		Bell
8	010	08	00001000	BS	&#008;		Back Space
9	011	09	00001001	HT			Horizontal Tab
10	012	0A	00001010	LF
	Line Feed
11	013	0B	00001011	VT	&#011;		Vertical Tab
12	014	0C	00001100	FF			Form Feed
13	015	0D	00001101	CR
	Carriage Return
14	016	0E	00001110	SO	&#014;		Shift Out / X-On
15	017	0F	00001111	SI	&#015;		Shift In / X-Off
16	020	10	00010000	DLE	&#016;		Data Line Escape
17	021	11	00010001	DC1	&#017;		Device Control 1 (oft. XON)
18	022	12	00010010	DC2	&#018;		Device Control 2
19	023	13	00010011	DC3	&#019;		Device Control 3 (oft. XOFF)
20	024	14	00010100	DC4	&#020;		Device Control 4
21	025	15	00010101	NAK	&#021;		Negative Acknowledgement
22	026	16	00010110	SYN	&#022;		Synchronous Idle
23	027	17	00010111	ETB	&#023;		End of Transmit Block
24	030	18	00011000	CAN	&#024;		Cancel
25	031	19	00011001	EM	&#025;		End of Medium
26	032	1A	00011010	SUB	&#026;		Substitute
27	033	1B	00011011	ESC	&#027;		Escape

28	034	1C	00011100	FS	&#028;		File Separator
29	035	1D	00011101	GS	&#029;		Group Separator
30	036	1E	00011110	RS	&#030;		Record Separator
31	037	1F	00011111	US	&#031;		Unit Separator

ASCII printable characters (character code 32-127)

Codes 32-127 are common for all the different variations of the ASCII table, they are called printable characters, represent letters, digits, punctuation marks, and a few miscellaneous symbols. You will find almost every character on your keyboard. Character 127 represents the command DEL.

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
32	040	20	00100000		&#32;		Space
33	041	21	00100001	!	&#33;		Exclamation mark
34	042	22	00100010	"	&#34;	&quot;	Double quotes (or speech marks)
35	043	23	00100011	#	&#35;		Number
36	044	24	00100100	$	&#36;		Dollar
37	045	25	00100101	%	&#37;		Procenttecken
38	046	26	00100110	&	&#38;	&amp;	Ampersand
39	047	27	00100111	'	&#39;		Single quote
40	050	28	00101000	(	&#40;		Open parenthesis (or open bracket)
41	051	29	00101001	)	&#41;		Close parenthesis (or close bracket)
42	052	2A	00101010	*	&#42;		Asterisk
43	053	2B	00101011	+	&#43;		Plus
44	054	2C	00101100	,	&#44;		Comma
45	055	2D	00101101	-	&#45;		Hyphen
46	056	2E	00101110	.	&#46;		Period, dot or full stop
47	057	2F	00101111	/	&#47;		Slash or divide
48	060	30	00110000	0	&#48;		Zero
49	061	31	00110001	1	&#49;		One
50	062	32	00110010	2	&#50;		Two
51	063	33	00110011	3	&#51;		Three
52	064	34	00110100	4	&#52;		Four
53	065	35	00110101	5	&#53;		Five
54	066	36	00110110	6	&#54;		Six

55	067	37	00110111	7	&#55;		Seven
56	070	38	00111000	8	&#56;		Eight
57	071	39	00111001	9	&#57;		Nine
58	072	3A	00111010	:	&#58;		Colon
59	073	3B	00111011	;	&#59;		Semicolon
60	074	3C	00111100	<	&#60;	&lt;	Less than (or open angled bracket)
61	075	3D	00111101	=	&#61;		Equals
62	076	3E	00111110	>	&#62;	&gt;	Greater than (or close angled bracket)
63	077	3F	00111111	?	&#63;		Question mark
64	100	40	01000000	@	&#64;		At symbol
65	101	41	01000001	A	&#65;		Uppercase A
66	102	42	01000010	B	&#66;		Uppercase B
67	103	43	01000011	C	&#67;		Uppercase C
68	104	44	01000100	D	&#68;		Uppercase D
69	105	45	01000101	E	&#69;		Uppercase E
70	106	46	01000110	F	&#70;		Uppercase F
71	107	47	01000111	G	&#71;		Uppercase G
72	110	48	01001000	H	&#72;		Uppercase H
73	111	49	01001001	I	&#73;		Uppercase I
74	112	4A	01001010	J	&#74;		Uppercase J
75	113	4B	01001011	K	&#75;		Uppercase K
76	114	4C	01001100	L	&#76;		Uppercase L
77	115	4D	01001101	M	&#77;		Uppercase M
78	116	4E	01001110	N	&#78;		Uppercase N
79	117	4F	01001111	O	&#79;		Uppercase O
80	120	50	01010000	P	&#80;		Uppercase P
81	121	51	01010001	Q	&#81;		Uppercase Q
82	122	52	01010010	R	&#82;		Uppercase R
83	123	53	01010011	S	&#83;		Uppercase S
84	124	54	01010100	T	&#84;		Uppercase T
85	125	55	01010101	U	&#85;		Uppercase U
86	126	56	01010110	V	&#86;		Uppercase V
87	127	57	01010111	W	&#87;		Uppercase W

88	130	58	01011000	X	&#88;	Uppercase X
89	131	59	01011001	Y	&#89;	Uppercase Y
90	132	5A	01011010	Z	&#90;	Uppercase Z
91	133	5B	01011011	[	&#91;	Opening bracket
92	134	5C	01011100	\	&#92;	Backslash
93	135	5D	01011101	]	&#93;	Closing bracket
94	136	5E	01011110	^	&#94;	Caret - circumflex
95	137	5F	01011111	_	&#95;	Underscore
96	140	60	01100000	`	&#96;	Grave accent
97	141	61	01100001	a	&#97;	Lowercase a
98	142	62	01100010	b	&#98;	Lowercase b
99	143	63	01100011	c	&#99;	Lowercase c
100	144	64	01100100	d	&#100;	Lowercase d
101	145	65	01100101	e	&#101;	Lowercase e
102	146	66	01100110	f	&#102;	Lowercase f
103	147	67	01100111	g	&#103;	Lowercase g
104	150	68	01101000	h	&#104;	Lowercase h
105	151	69	01101001	i	&#105;	Lowercase i
106	152	6A	01101010	j	&#106;	Lowercase j
107	153	6B	01101011	k	&#107;	Lowercase k
108	154	6C	01101100	l	&#108;	Lowercase l
109	155	6D	01101101	m	&#109;	Lowercase m
110	156	6E	01101110	n	&#110;	Lowercase n
111	157	6F	01101111	o	&#111;	Lowercase o
112	160	70	01110000	p	&#112;	Lowercase p
113	161	71	01110001	q	&#113;	Lowercase q
114	162	72	01110010	r	&#114;	Lowercase r
115	163	73	01110011	s	&#115;	Lowercase s
116	164	74	01110100	t	&#116;	Lowercase t
117	165	75	01110101	u	&#117;	Lowercase u
118	166	76	01110110	v	&#118;	Lowercase v
119	167	77	01110111	w	&#119;	Lowercase w
120	170	78	01111000	x	&#120;	Lowercase x

121	171	79	01111001	y	&#121;		Lowercase y
122	172	7A	01111010	z	&#122;		Lowercase z
123	173	7B	01111011	{	&#123;		Opening brace
124	174	7C	01111100	|	&#124;		Vertical bar
125	175	7D	01111101	}	&#125;		Closing brace
126	176	7E	01111110	~	&#126;		Equivalency sign - tilde
127	177	7F	01111111		&#127;		Delete

The extended ASCII codes (character code 128-255)

There are several different variations of the 8-bit ASCII table. The table below is according to ISO 8859-1, also called ISO Latin-1. Codes 129-159 contain the Microsoft® Windows Latin-1 extended characters.

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
128	200	80	10000000	€	&#128;	&euro;	Euro sign
129	201	81	10000001				
130	202	82	10000010	‚	&#130;	&sbquo;	Single low-9 quotation mark
131	203	83	10000011	*ƒ*	&#131;	&fnof;	Latin small letter f with hook
132	204	84	10000100	„	&#132;	&bdquo;	Double low-9 quotation mark
133	205	85	10000101	…	&#133;	&hellip;	Horizontal ellipsis
134	206	86	10000110	†	&#134;	&dagger;	Dagger
135	207	87	10000111	‡	&#135;	&Dagger;	Double dagger
136	210	88	10001000	ˆ	&#136;	&circ;	Modifier letter circumflex accent
137	211	89	10001001	‰	&#137;	&permil;	Per mille sign
138	212	8A	10001010	Š	&#138;	&Scaron;	Latin capital letter S with caron
139	213	8B	10001011	‹	&#139;	&lsaquo;	Single left-pointing angle quotation
140	214	8C	10001100	Œ	&#140;	&OElig;	Latin capital ligature OE
141	215	8D	10001101				
142	216	8E	10001110	Ž	&#142;		Latin captial letter Z with caron
143	217	8F	10001111				
144	220	90	10010000				
145	221	91	10010001	‘	&#145;	&lsquo;	Left single quotation mark
146	222	92	10010010	’	&#146;	&rsquo;	Right single quotation mark
147	223	93	10010011	“	&#147;	&ldquo;	Left double quotation mark
148	224	94	10010100	”	&#148;	&rdquo;	Right double quotation mark

149	225	95	10010101	•	&#149;	&bull;	Bullet
150	226	96	10010110	–	&#150;	&ndash;	En dash
151	227	97	10010111	—	&#151;	&mdash;	Em dash
152	230	98	10011000	˜	&#152;	&tilde;	Small tilde
153	231	99	10011001	™	&#153;	&trade;	Trade mark sign
154	232	9A	10011010	š	&#154;	&scaron;	Latin small letter S with caron
155	233	9B	10011011	›	&#155;	&rsaquo;	Single right-pointing angle quotation mark
156	234	9C	10011100	œ	&#156;	&oelig;	Latin small ligature oe
157	235	9D	10011101				
158	236	9E	10011110	ž	&#158;		Latin small letter z with caron
159	237	9F	10011111	Ÿ	&#159;	&yuml;	Latin capital letter Y with diaeresis
160	240	A0	10100000				Non-breaking space
161	241	A1	10100001	¡	&#161;	&iexcl;	Inverted exclamation mark
162	242	A2	10100010	¢	&#162;	&cent;	Cent sign
163	243	A3	10100011	£	&#163;	&pound;	Pound sign
164	244	A4	10100100	¤	&#164;	&curren;	Currency sign
165	245	A5	10100101	¥	&#165;	&yen;	Yen sign
166	246	A6	10100110	¦	&#166;	&brvbar;	Pipe, Broken vertical bar
167	247	A7	10100111	§	&#167;	&sect;	Section sign
168	250	A8	10101000	¨	&#168;	&uml;	Spacing diaeresis - umlaut
169	251	A9	10101001	©	&#169;	&copy;	Copyright sign
170	252	AA	10101010	ª	&#170;	&ordf;	Feminine ordinal indicator
171	253	AB	10101011	«	&#171;	&laquo;	Left double angle quotes
172	254	AC	10101100	¬	&#172;	&not;	Not sign
173	255	AD	10101101		&#173;	&shy;	Soft hyphen
174	256	AE	10101110	®	&#174;	&reg;	Registered trade mark sign
175	257	AF	10101111	¯	&#175;	&macr;	Spacing macron - overline
176	260	B0	10110000	°	&#176;	&deg;	Degree sign
177	261	B1	10110001	±	&#177;	&plusmn;	Plus-or-minus sign
178	262	B2	10110010	²	&#178;	&sup2;	Superscript two - squared
179	263	B3	10110011	³	&#179;	&sup3;	Superscript three - cubed
180	264	B4	10110100	´	&#180;	&acute;	Acute accent - spacing acute
181	265	B5	10110101	µ	&#181;	&micro;	Micro sign

182	266	B6	10110110	¶	&#182;	&para;	Pilcrow sign - paragraph sign
183	267	B7	10110111	·	&#183;	&middot;	Middle dot - Georgian comma
184	270	B8	10111000	¸	&#184;	&cedil;	Spacing cedilla
185	271	B9	10111001	¹	&#185;	&sup1;	Superscript one
186	272	BA	10111010	º	&#186;	&ordm;	Masculine ordinal indicator
187	273	BB	10111011	»	&#187;	&raquo;	Right double angle quotes
188	274	BC	10111100	¼	&#188;	&frac14;	Fraction one quarter
189	275	BD	10111101	½	&#189;	&frac12;	Fraction one half
190	276	BE	10111110	¾	&#190;	&frac34;	Fraction three quarters
191	277	BF	10111111	¿	&#191;	&iquest;	Inverted question mark
192	300	C0	11000000	À	&#192;	&Agrave;	Latin capital letter A with grave
193	301	C1	11000001	Á	&#193;	&Aacute;	Latin capital letter A with acute
194	302	C2	11000010	Â	&#194;	&Acirc;	Latin capital letter A with circumflex
195	303	C3	11000011	Ã	&#195;	&Atilde;	Latin capital letter A with tilde
196	304	C4	11000100	Ä	&#196;	&Auml;	Latin capital letter A with diaeresis
197	305	C5	11000101	Å	&#197;	&Aring;	Latin capital letter A with ring above
198	306	C6	11000110	Æ	&#198;	&AElig;	Latin capital letter AE
199	307	C7	11000111	Ç	&#199;	&Ccedil;	Latin capital letter C with cedilla
200	310	C8	11001000	È	&#200;	&Egrave;	Latin capital letter E with grave
201	311	C9	11001001	É	&#201;	&Eacute;	Latin capital letter E with acute
202	312	CA	11001010	Ê	&#202;	&Ecirc;	Latin capital letter E with circumflex
203	313	CB	11001011	Ë	&#203;	&Euml;	Latin capital letter E with diaeresis
204	314	CC	11001100	Ì	&#204;	&Igrave;	Latin capital letter I with grave
205	315	CD	11001101	Í	&#205;	&Iacute;	Latin capital letter I with acute
206	316	CE	11001110	Î	&#206;	&Icirc;	Latin capital letter I with circumflex
207	317	CF	11001111	Ï	&#207;	&Iuml;	Latin capital letter I with diaeresis
208	320	D0	11010000	Ð	&#208;	&ETH;	Latin capital letter ETH
209	321	D1	11010001	Ñ	&#209;	&Ntilde;	Latin capital letter N with tilde
210	322	D2	11010010	Ò	&#210;	&Ograve;	Latin capital letter O with grave
211	323	D3	11010011	Ó	&#211;	&Oacute;	Latin capital letter O with acute
212	324	D4	11010100	Ô	&#212;	&Ocirc;	Latin capital letter O with circumflex
213	325	D5	11010101	Õ	&#213;	&Otilde;	Latin capital letter O with tilde
214	326	D6	11010110	Ö	&#214;	&Ouml;	Latin capital letter O with diaeresis

215	327	D7	11010111	×	×	×	Multiplication sign
216	330	D8	11011000	Ø	Ø	Ø	Latin capital letter O with slash
217	331	D9	11011001	Ù	Ù	Ù	Latin capital letter U with grave
218	332	DA	11011010	Ú	Ú	Ú	Latin capital letter U with acute
219	333	DB	11011011	Û	Û	Û	Latin capital letter U with circumflex
220	334	DC	11011100	Ü	Ü	Ü	Latin capital letter U with diaeresis
221	335	DD	11011101	Ý	Ý	Ý	Latin capital letter Y with acute
222	336	DE	11011110	Þ	Þ	Þ	Latin capital letter THORN
223	337	DF	11011111	ß	ß	ß	Latin small letter sharp s - ess-zed
224	340	E0	11100000	à	à	à	Latin small letter a with grave
225	341	E1	11100001	á	á	á	Latin small letter a with acute
226	342	E2	11100010	â	â	â	Latin small letter a with circumflex
227	343	E3	11100011	ã	ã	ã	Latin small letter a with tilde
228	344	E4	11100100	ä	ä	ä	Latin small letter a with diaeresis
229	345	E5	11100101	å	å	å	Latin small letter a with ring above
230	346	E6	11100110	æ	æ	æ	Latin small letter ae
231	347	E7	11100111	ç	ç	ç	Latin small letter c with cedilla
232	350	E8	11101000	è	è	è	Latin small letter e with grave
233	351	E9	11101001	é	é	é	Latin small letter e with acute
234	352	EA	11101010	ê	ê	ê	Latin small letter e with circumflex
235	353	EB	11101011	ë	ë	ë	Latin small letter e with diaeresis
236	354	EC	11101100	ì	ì	ì	Latin small letter i with grave
237	355	ED	11101101	í	í	í	Latin small letter i with acute
238	356	EE	11101110	î	î	î	Latin small letter i with circumflex
239	357	EF	11101111	ï	ï	ï	Latin small letter i with diaeresis
240	360	F0	11110000	ð	ð	ð	Latin small letter eth
241	361	F1	11110001	ñ	ñ	ñ	Latin small letter n with tilde
242	362	F2	11110010	ò	ò	ò	Latin small letter o with grave
243	363	F3	11110011	ó	ó	ó	Latin small letter o with acute
244	364	F4	11110100	ô	ô	ô	Latin small letter o with circumflex
245	365	F5	11110101	õ	õ	õ	Latin small letter o with tilde
246	366	F6	11110110	ö	ö	ö	Latin small letter o with diaeresis
247	367	F7	11110111	÷	÷	÷	Division sign

248	370	F8	11111000	ø	&#248;	&oslash;	Latin small letter o with slash
249	371	F9	11111001	ù	&#249;	&ugrave;	Latin small letter u with grave
250	372	FA	11111010	ú	&#250;	&uacute;	Latin small letter u with acute
251	373	FB	11111011	û	&#251;	&ucirc;	Latin small letter u with circumflex
252	374	FC	11111100	ü	&#252;	&uuml;	Latin small letter u with diaeresis
253	375	FD	11111101	ý	&#253;	&yacute;	Latin small letter y with acute
254	376	FE	11111110	þ	&#254;	&thorn;	Latin small letter thorn
255	377	FF	11111111	ÿ	&#255;	&yuml;	Latin small letter y with diaeresis

Appendix B: Self-Written C/C++ Codes (Except the A/U-Law algorithms)

```
#ifndef _GLOBALS_H_
#define _GLOBALS_H_
// To make our codes more cross-platform and without the need to continuously
// rewrite our codes for different platforms, we include this local inttypes
// header to define the standard fix-width data types part of the C99 standard.

// The ISO/IEC 9899:1999 (E), or C99 standard, defines the prototypes for
// 'Format conversion of integer types' on page 197 (page 211 in pdf).
// The implementation for these 'Format conversion of integer types' is
// defined on page 254 (or page 268 of the pdf).

// Microsoft VC compilers, which are based on the C++0x language specification,
// does not support these standard types. These types were later defined. Microsoft
// selectively added support for particular C++11 language specifications, however
// these standard integer types were not the chosen few features Microsoft implemented.
// Therefore, we include this header, which fills in the necessary gap for us to use
// these types by defining them. This header does not make the VC compiler completely
// compatible with the C99 or C++11 standards, however, it will do for this project.

// Reference documents:
// You can find ISO/IEC 9899:1999 (E), aka C99 at:
// http://cs.nyu.edu/courses/spring13/CSCI-GA.2110-001/downloads/C99.pdf

// You can find the ISO/IEC 14882:2003(E), aka C++11 at:
// http://cs.nyu.edu/courses/spring13/CSCI-GA.2110-
001/downloads/C++%20Standard%202003.pdf

// Additional references:
// 1. http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx
// 2. https://www2.research.att.com/~bs/what-is-2009.pdf
#include "inttypes.h"

// We include our standard input/output library. I like 'printf'
// and of course for our file input/output data structures/algorithms
#include <stdio.h>
#include <stdlib.h>

#include <SDL/SDL.h>
#include <SDL/SDL_audio.h>
#include <SDL/SDL_keyboard.h>

typedef struct WaveInfo {
    // The WAVE file format is a subset of the RIFF, which is an extension of IFF.
    // The IFF specification defines the concept and structure of a chunk.
    // This chunk is contains a 4-byte sequence, a 32-bit 'chunk size',
    // and a 'data' section of size 'chunk size'.
    // See http://www.digitalpreservation.gov/formats/fdd/fdd000115.shtml
    // Document link: http://www.martinreddy.net/gfx/2d/IFF.txt

    // However, Wave is a subset of RIFF, which is an extension of IFF.
    // RIFF extends the 'data' section of IFF to, optionally, include
    // sub chunks. Therefore, RIFF documents can contain nested chunks
    // However, Only the RIFF and LIST chunk types can contain other
    // chunks. All other chunks are defined precisely as in IFF That is,
    // their data section does not contain other chunks.
```

```
 // Document link: http://www.kk.iij4u.or.jp/~kondo/wave/mpidata.txt

/* According to RIFF's specification (2-2):
-----------------------------------------------------------
'We can represent a chunk with the following notation
 (in this example, the ckSize and pad byte are
 implicit):RIFF chunk;notation

  <ckID> ( <ckData> )

 Two types of chunks, the LIST and RIFF chunks, may
 contain nested chunks, or subchunks. These special
 chunk types are discussed later in this document. All
 other chunk types store a single element of binary data
 in <ckData>.'

 The basic building block of a RIFF file is called a
 chunk. Using C syntax, a chunk can be defined as
 follows:RIFF chunk;defined in C syntax

 typedef unsigned long DWORD;
 typedef unsigned char BYTE;

 typedef DWORD FOURCC;   // Four-character code

 typedef FOURCC CKID;    // Four-character-code chunk identifier
 typedef DWORD CKSIZE;   // 32-bit unsigned size value

 -----------------------------------------------------
 Part      Description
 -----------------------------------------------------
 ckID      A [four-character code] that identifies the
           representation of the chunk data data. A
           program reading a RIFF file can skip over
           any chunk whose chunk ID it doesn't
           recognize; it simply skips the number of
           bytes specified by ckSize plus the pad
           byte, if present.

 ckSize    A 32-bit unsigned value identifying the
           size of ckData. This size value does not
           include the size of the ckID or ckSize
           fields or the pad byte at the end of
           ckData.
 */

 // Beginning of our RIFF chunk. All other chunks are nested in this one
 uint32_t    riff_chunk_id; // Chunk id is an unsigned 4-bytes character code
 uint32_t    riff_chunk_size; //  Chuck size is an unsigned 32-bit integer

 /* Data section of RIFF Chunk. First we define the format of our RIFF.
    Then we define all the necessary chunks for our RIFF format. */
 uint32_t    riff_format; // This identifies what sort of RIFF file we have.

 // Wave is defined in the RIFF specifications has consisting
 // of two sub chunks. The two sub chunks are 'fmt' and 'data'
 // The 'fmt' chunk describes the properties of our audio/data.
 uint32_t    fmt_chunk_id;
```

```
uint32_t    fmt_chunk_size;

/*
    *NOTE on BNF definitions/convention*
    -----------------------------------
    Elements on a new line do not represent 'alternatives'. If
    'alternatives' is meant, then a '|' (OR) operator will
    separate the alterative fields. Therefore:

    <item>: <ckID> (
                        <item_A>
                        <item_B>
                    }

    is the same as:

    <item>: <ckID> ( item_A> <item_B> )

    Therefore, a new line does not represent 'alternatives'.
    Separating our items in new lines represents concatenation
    with a single space between the items. To express
    'alternatives', the following conventions are adopted:

    <item>: <ckID> (
                        <item_A> |
                        <item_B>
                     }

    This can also be represented in a single line as:

    <item>: <ckID> ( <item_A> | <item_B> )

Remember from above, the items between the opening '(' and
closing ')' represent the chunk data. The 'chunk size' field is
always implicitly implied with this notation, as stated by RIFF.

RIFF defines the 'fmt', or format, chunk as the following (BNF definition):
<fmt-ck>: fmt (
                <common-fields>
                <format-specific-fields>
              )

<common-fields>:
      struct
        {
           WORD     wFormatTag;         // Format category (format of audio)
           WORD     wChannels;          // Number of channels
           DWORD    dwSamplesPerSec;    // Sampling rate
           DWORD    dwAvgBytesPerSec;   // For buffer estimation
           WORD     wBlockAlign;        // Data block size
        }

*IMPORTANT NOTE I WILL ADD*
-------------------------------------------------
WORD    = unsigned 2-bytes --> unsigned 16-bits;
DWORD   = unsigned 4-bytes --> unsigned 32-bits;
-------------------------------------------------
```

```
the <format-specific-fields> depends on the wFormatTag we are using.
According to the specificiation document:
------------------------------------------------------------------
'The format category of a WAVE file is specified by the
 value of the wFormatTag field of the fmt chunk. The
 representation of data in <wave-data>, and the content
 of the <format-specific-fields> of the fmt chunk,
 depend on the format category.Waveform files;format
 categories

 The currently defined open non-proprietary WAVE format
 categories are as follows:


 wFormatTag Value           Format Category
 WAVE_FORMAT_PCM (0x0001)   Microsoft Pulse Code Modulation (PCM) format


 The following are the registered proprietary WAVE format categories:

 wFormatTag Value         Format Category
 IBM_FORMAT_MULAW         IBM mu-law format
 (0x0101)
 IBM_FORMAT_ALAW (0x0102) IBM a-law format
 IBM_FORMAT_ADPCM         IBM AVC Adaptive
 (0x0103)                 Differential Pulse Code
                          Modulation format'
------------------------------------------------------------------

In our case, we will only support WAVE_FORMAT_PCM, or PCM. Therefore,
we will expect the 32-bit wFormatTag field to take on the value 0x00000001.
Because 32-bit integers are usually stored in Big Endian, we expect the
value 0x10000000 for PCM wave in the actual file.

Also, if our wFormatTag is PCM, then the <format-specific-fields> (see above)
will have 1 additional field, 'Bits per sample', Therefore, we need to add 1
field to our <format-specific-fields> section. As the document states:

------------------------------------------------------------------
'The following sections describe the Microsoft
 WAVE_FORMAT_PCM format.


 Pulse Code Modulation (PCM) Format

 If the wFormatTag field of the <fmt-ck> is set to
 WAVE_FORMAT_PCM, then the waveform data consists of
 samples represented in pulse code modulation (PCM)
 format. For PCM waveform data, the <format-specific-
 fields> is defined as follows:

  <PCM-format-specific>:
        struct
        {
           WORD wBitsPerSample;      // Sample size
        }

 The wBitsPerSample field specifies the number of bits
```

```
     of data used to represent each sample of each channel.
     If there are multiple channels, the sample size is the
     same for each channel.'
     ---------------------------------------------------------------------

     Therefore, our format chunk definition expands to:

     <fmt-ck>: fmt (
                    <common-fields>
                    <format-specific-fields>
                  )

     <common-fields>:
            struct
            {
               WORD     wFormatTag;         // Format category (format of audio)
               WORD     wChannels;          // Number of channels
               DWORD    dwSamplesPerSec;    // Sampling rate
               DWORD    dwAvgBytesPerSec;   // For buffer estimation
               WORD     wBlockAlign;        // Data block size
            }

    <format-specific-fields>:
            struct
            {
               WORD wBitsPerSample;         // Sample size
            }

    *IMPORTANT NOTE I WILL ADD #2*
    -------------------------------------------------
    WORD    = unsigned 2-bytes --> uint16_t;
    DWORD   = unsigned 4-bytes --> uint32_t;
    -------------------------------------------------
    */

    // -------------------------------------------------
    // + Implementation of the Format Chunk's fields    +
    // + Note we prefix our variables with the size     +
    // + of the data type. For example, wFormatTag      +
    // + states that 'FormatTag' is of size 'WORD' and  +
    // + in 'dwSamplesPerSec', 'dw' stands for 'DWORD'. +
    // + The types are defined in the notes above.      +
    // -------------------------------------------------
    // <common-fields>:
    uint16_t wFormatTag;        // Format category (PCM = 1)
    uint16_t wChannels;         // Number of channels
    uint32_t dwSamplesPerSec;   // Sampling rate
    uint32_t dwAvgBytesPerSec;  // For buffer estimation (average bytes/sec)
    uint16_t wBlockAlign;       // Data block size

    // <format-specific-fields>
    uint16_t wBitsPerSample;    // Sample size

    // We may have these two additional fields, particularly if wFormatTag is not PCM.
    uint16_t wExtraParamSize;   // Extra parameter size (used by some non-PCM audio
formats)
    uint8_t *extraParams;       // We wont know what the size of this array is until
later
```

```
/*
Our format chunk is defined as:
-------------------------------------------------------------
<fmt-ck>: fmt (
                  <common-fields>
                  <format-specific-fields>
              )
-------------------------------------------------------------
We have all the necessary fields, so we move onto the second
field the WAVE RIFF format defines, the 'data' chunk.

The 'data' chunk of a WAVE file is defined as:
-------------------------------------------------------------
'The <wave-data> contains the waveform data. It is
defined as follows:

 <wave-data>:   { <data-ck> | <wave-list> }

 // NOTE: The specification replaces '<wave-list>' above
 // with '<data-list>'. However, I believe that is a typo,
 // so I am following what I assume is correct and was meant.

 <data-ck>:     data ( <wave-data> )

 <wave-list>:   LIST (
                        'wavl' { <data-ck> | <silence-ck> ... }
                    )

  <silence-ck>: slnt ( <dwSamples:DWORD> ) // Count of silent samples

 Note:  The slnt chunk represents silence, not
 necessarily a repeated zero volume or baseline sample.
 In 16-bit PCM data, if the last sample value played
 before the silence section is a 10000, then if data is
 still output to the D to A converter, it must maintain
 the 10000 value. If a zero value is used, a click may
 be heard at the start and end of the silence section.
 If play begins at a silence section, then a zero value
 might be used since no other information is available.
 A click might be created if the data following the
 silent section starts with a nonzero value.'
-------------------------------------------------------------

According to the definition above, the 'wave data' section is
a sequence of 'data-ck' or 'wave-list' chunks. As the specification
stated earlier, the only chunks that contain other chunks are
the 'RIFF' and 'LIST' chunks.

The <wave-list> chunk is a 'LIST' chunk whose data section
contains a sequence of <data-ck> or <silence-ck>. The silence
tells us that no sound is to be played for '<dwSamples:DWORD>'
number of audio samples (or data samples). Therefore, if our
data section is a <wave-list> chunk which contains alternating
sequence of <data-ck> and <silence-ck>, then we will know that
whenever we encounter a <silence-ck>, we must send 0 (zero)
amplitude data to the speaker for the next <dwSamples:DWORD>
sample counts. Alternatively, we can stop all processing for
```

```
x number of seconds that corresponds to the processing/playback
of <dwSamples:DWORD> number of sample counts. Therefore, <silence-ck>
can be used to compress wave files because we would represent
<dwSamples:DWORD> number of samples as a single 32-bit type, as
opposed to <wBitsPerSample:WORD> (see above) number of bits.
However, many decoders skip this optional <silence-ck>, therefore
we will also skip it. Since we wont support <silence-ck>, there
wont be a need to add support for <wave-list> chunks.

The <data-ck>, on the other hand, is not a 'LIST' chunk.
Therefore, it's data section will not contain other chunks.
It's data section will contain the data defined by the 'fmt'
chunk. This is the only part of the <wave-data> definition
we will support because it is the only chunk type supported
by most wave codecs.

Depending on the <wFormatTag:WORD>, <wChannels:WORD>, and
<wBitsPerSample:WORD>, values defined in the <fmt-ck>, we
could have various layouts. <wBitsPerSample:WORD> determines
how many bits we use per channel, and the <wChannels:WORD>
determines how many channels do we have per audio sample.

Some examples of layouts we can have for PCM are:
*NOTE: Base 10, or decimal, is being used for these examples*

-------------------
+ PCM_DATA_LAYOUT +
-------------------

(8-bit Mono):
--------------------------------------------
<wFormatTag:WORD>       =   1 (PCM)
<wBitsPerSample:WORD>   =   8
<wChannels:WORD>        =   1

Sample  1   Sample  2   Sample  3   Sample  4
Channel 0   Channel 0   Channel 0   Channel 0

Where each 'Channel 0' is 8-bits.


(8-bit Stereo):
--------------------------------------------
<wFormatTag:WORD>       =   1 (PCM)
<wBitsPerSample:WORD>   =   8
<wChannels:WORD>        =   2

     Sample  1            Sample  2             Sample  3
Channel 0 Channel 1   Channel 0 Channel 1   Channel 0 Channel 1

Where each 'Channel 0' is 8-bits and each 'Channel 1' is 8-bits.


(16-bit Mono):
--------------------------------------------
<wFormatTag:WORD>       =   1 (PCM)
<wBitsPerSample:WORD>   =   16
<wChannels:WORD>        =   1
```

```
     Sample  1            Sample  2             Sample  3
Channel 0 Channel 0   Channel 0 Channel 0   Channel 0 Channel 0

Where each 'Channel 0' is 8-bits, therefore, 'Channel 0' = 16-bits.


(16-bit Stereo):
-----------------------------------------
<wFormatTag:WORD>       =   1 (PCM)
<wBitsPerSample:WORD>   =   16
<wChannels:WORD>        =   2

               Sample  1                                Sample  2
Channel 0 Channel 0 Channel 1 Channel 1   Channel 0 Channel 0 Channel 1 Channel 1

Where each 'Channel 0' is 8-bits and each 'Channel 1' is 8-bits.
Therefore, 'Channel 0' = 16-bits and 'Channel 1' = 16-bits.

Since we wont support for <wave-list>, the definition
of what we will implement will change from:

*Standard definition*
-------------------------------------------------------------------
<wave-data>:   { <data-ck> | <wave-list> }

<data-ck>:     data ( <wave-data> | <wave-list> )

<wave-list>:   LIST (
                        'wavl' { <data-ck> | <silence-ck> ... }
                     )

<silence-ck>: slnt ( <dwSamples:DWORD> ) // Count of silent samples
-------------------------------------------------------------------

will change to our implementation definition:

*Our Implementation definition*
-------------------------------------------------------------------
<wave-data>:   { <data-ck> }

<data-ck>:     data ( PCM_DATA_LAYOUT )
-------------------------------------------------------------------
*/

// ---------------------------------------------------
// + Implementation of the Data Chunk's fields    +
// + Note we prefix our variables with the size     +
// + of the data type. For example, wFormatTag      +
// + states that 'FormatTag' is of size 'WORD' and  +
// + in 'dwSamplesPerSec', 'dw' stands for 'DWORD'. +
// + The types are defined in the note below.      +
// ---------------------------------------------------
// *IMPORTANT NOTE*
// ---------------------------------------------------
// WORD    = unsigned 2-bytes --> uint16_t;
// DWORD   = unsigned 4-bytes --> uint32_t;
// ---------------------------------------------------
```

```
    // <data-ck>:
    uint32_t    data_chunk_id;
    uint32_t    data_chunk_size;

    // Because we cannot determine <wFormatTag:WORD>, <wChannels:WORD>,
    // and <wBitsPerSample:WORD> until we read some actual data and
    // these fields can change, we will implement this type as a 2D
    // array. However, we also do not know the array size, therefore,
    // we cannot directly declare the type as an array. We will declare
    // it as a pointer to an unsigned 8-bit integer pointer. Therefore,
    // the data field will be an uint8_t pointer to an unit8_t pointer.
    // We will later allocate the rows and columns of our pointer
    // structure, once we know what the size is. We are using an unsigned
    // 8-bit pointer to a pointer because each channel can be expressed in
    // units of 8 bits. Therefore, if we have a Stereo channel, then that will
    // consume two 8-bit columns in our 2D array. Therefore, the columns of
    // our 2D array will contain the channels of our sample and each row
    // will contain another sample.

    void *data; // 2D array that will contain the WAVE's data

    int endian; // endian of our document (depends on RIFF or RIFX)

} WaveInfo;

#define LITTLE_ENDIAN   0
#define BIG_ENDIAN      1

const int cpu_endian_test = 1;
#define cpu_endian() ((*(char*)&cpu_endian_test) == 0? BIG_ENDIAN : LITTLE_ENDIAN)

enum RIFF_INFO {
    RIFF    = 0x52494646,
    RIFX    = 0x52494658,
    WAVE    = 0x57415645,
    FMT     = 0x666D7420,
    DATA    = 0x64617461,


    WAVE_FORMAT_PCM     = 0x0001,
    //IBM_FORMAT_MULAW    = 0x0101,
    IBM_FORMAT_MULAW    = 0x0007,
    IBM_FORMAT_ALAW     = 0x0102,
    IBM_FORMAT_ADPCM    = 0x0103
};

void playWave(WaveInfo *wInfo);
void changeSampleRate(WaveInfo *wInfo, uint32_t new_rate);

void  alaw_expand(long lseg, short *logbuf, short *linbuf);
void  ulaw_expand(long lseg, short *logbuf, short *linbuf);

#endif
```

```
#include "globals.h"

/* ................... Begin of alaw_expand() ..................... */
/*
  ============================================================================

   FUNCTION NAME: alaw_expand

   DESCRIPTION: ALaw decoding rule according ITU-T Rec. G.711.

   PROTOTYPE: void alaw_expand(long lseg, short *logbuf, short *linbuf)

   PARAMETERS:
     lseg:    (In)  number of samples
     logbuf:  (In)  buffer with compressed samples (8 bit right justified,
                      without sign extension)
     linbuf:  (Out) buffer with linear samples (13 bits left justified)

   RETURN VALUE: none.

   HISTORY:
   10.Dec.91  1.0    Separated A-law expansion function

  ============================================================================
*/
void            alaw_expand(long lseg, short *logbuf, short *linbuf)
{
  short           ix, mant, iexp;
  long            n;

  for (n = 0; n < lseg; n++)
  {
    ix = logbuf[n] ^ (0x0055);     /* re-toggle toggled bits */

    ix &= (0x007F);         /* remove sign bit */
    iexp = ix >> 4;         /* extract exponent */
    mant = ix & (0x000F);   /* now get mantissa */
    if (iexp > 0)
      mant = mant + 16;           /* add leading '1', if exponent > 0 */

    mant = (mant << 4) + (0x0008); /* now mantissa left justified and */
    /* 1/2 quantization step added */
    if (iexp > 1)           /* now left shift according exponent */
      mant = mant << (iexp - 1);

    linbuf[n] = logbuf[n] > 127    /* invert, if negative sample */
      ? mant
      : -mant;
  }
}
/* ................... End of alaw_expand() ..................... */

/* ................... Begin of ulaw_expand() ..................... */
/*
  ============================================================================

   FUNCTION NAME: ulaw_expand
```

```
   DESCRIPTION: Mu law decoding rule according ITU-T Rec. G.711.

   PROTOTYPE: void ulaw_expand(long lseg, short *logbuf, short *linbuf)

   PARAMETERS:
     lseg:    (In)  number of samples
     logbuf:  (In)  buffer with compressed samples (8 bit right justified,
                      without sign extension)
     linbuf:  (Out) buffer with linear samples (14 bits left justified)

   RETURN VALUE: none.

   HISTORY:
   10.Dec.91  1.0    Separated mu law expansion function

  ============================================================================
*/

void ulaw_expand(long lseg, short *logbuf, short *linbuf)
{
  long            n;        /* aux.var. */
  short           segment;  /* segment (Table 2/G711, column 1) */
  short           mantissa; /* low  nibble of log companded sample */
  short           exponent; /* high nibble of log companded sample */
  short           sign;           /* sign of output sample */
  short           step;

  for (n = 0; n < lseg; n++)
  {
    sign = logbuf[n] < (0x0080) /* sign-bit = 1 for positiv values */
        ? -1 : 1;
    mantissa = ~logbuf[n];  /* 1's complement of input value */
    exponent = (mantissa >> 4) & (0x0007);      /* extract exponent */
    segment = exponent + 1; /* compute segment number */
    mantissa = mantissa & (0x000F);       /* extract mantissa */

    /* Compute Quantized Sample (14 bit left justified!) */
    step = (4) << segment;  /* position of the LSB */
    /* = 1 quantization step) */
    linbuf[n] = sign *            /* sign */
      (((0x0080) << exponent)     /* '1', preceding the mantissa */
       + step * mantissa    /* left shift of mantissa */
       + step / 2           /* 1/2 quantization step */
       - 4 * 33
      );
  }
}
/* ................... End of ulaw_expand() ..................... */

void cpu_uint16(uint16_t &val, int val_endian)
{
    if(cpu_endian() == val_endian)
        return;

    val = val << 8
        | val >> 8 & 0x00FF;
}
```

```
void cpu_uint32(uint32_t &val, int val_endian)
{
    if(cpu_endian() == val_endian)
        return;

    val = val << 24
        | val << 8  & 0x00FF0000
        | val >> 8  & 0x0000FF00
        | val >> 24 & 0x000000FF;
}

int read_bytes(uint8_t *dest_data, uint32_t num_bytes, FILE *file) {
       return fread(dest_data, 1, num_bytes, file);;
}

int read_bytes_short(short *dest_data, uint32_t num_bytes, FILE *file) {
       return fread(dest_data, 2, num_bytes, file);;
}

int read_uint8(uint8_t &uint8_val, FILE *file) {
       return fread(&uint8_val, 1, 1, file);;
}

int read_uint16(uint16_t &uint16_val, FILE *file) {
       return fread(&uint16_val, 2, 1, file);
}

int read_uint32(uint32_t &uint32_val, FILE *file) {
       return fread(&uint32_val, 4, 1, file);
}

int skip_bytes(uint32_t num_bytes, FILE *file)
{
    uint8_t *buf;
    buf = (uint8_t *)malloc(num_bytes);

    if(buf == NULL)
        return -1;

    return fread(buf, 1, num_bytes, file);
}

void changeSampleRate(WaveInfo *wInfo, uint32_t new_rate)
{
    wInfo->dwSamplesPerSec = new_rate;
    wInfo->dwAvgBytesPerSec = wInfo->dwSamplesPerSec*wInfo->wChannels*wInfo-
>wBitsPerSample/8;
}

int decodeWave(char *filename, WaveInfo *wInfo)
{
    FILE *file;
    uint8_t byte;
    uint16_t word;
    uint32_t dword;

    uint32_t bytes_remaining;
```

```
file = fopen(filename, "rb");
if(file == NULL) {
    printf("File not found or could not be opened!\n");
    return -1;
}

while(read_uint32(dword, file) > 0) {
    cpu_uint32(dword, BIG_ENDIAN);
    switch(dword) {
    case RIFF:
    case RIFX:
        wInfo->riff_chunk_id = dword;
        if(wInfo->riff_chunk_id == RIFF) {
            wInfo->endian = LITTLE_ENDIAN;
            printf("<riff-ck>:\n");
        } else {
            wInfo->endian = BIG_ENDIAN;
            printf("<rifx-ck>:\n");
        }

        // Process the chunk size
        read_uint32(dword, file);
        cpu_uint32(dword, wInfo->endian);
        wInfo->riff_chunk_size = dword;
        printf("chunk size: %08X\n", wInfo->riff_chunk_size);

        // Process the 'RIFF or RIFX' chunk format/type
        // this should be 'WAVE'. If it is not, we error.
        read_uint32(dword, file);
        cpu_uint32(dword, BIG_ENDIAN);
        wInfo->riff_format = dword;
        if(wInfo->riff_format == WAVE)
            printf("We have a WAVE file!\n\n");
        else
            printf("We do not have a WAVE file!\n\n");
        break;
    case FMT:
        printf("<fmt-ck>:\n");

        // Process the chunk size
        read_uint32(dword, file);
        cpu_uint32(dword, wInfo->endian);
        wInfo->fmt_chunk_size = dword;
        bytes_remaining = wInfo->fmt_chunk_size;
        printf("chunk size: %08X\n", wInfo->fmt_chunk_size);

        // Process audio format
        read_uint16(word, file);
        cpu_uint16(word, wInfo->endian);
        wInfo->wFormatTag = word;
        bytes_remaining -= 2;
        printf("Audio format: %u\n", wInfo->wFormatTag);

        // Process number of channels
        read_uint16(word, file);
        cpu_uint16(word, wInfo->endian);
        wInfo->wChannels = word;
        bytes_remaining -= 2;
```

```
            printf("Channels: %u\n", wInfo->wChannels);

            // Process the sample rate
            read_uint32(dword, file);
            cpu_uint32(dword, wInfo->endian);
            wInfo->dwSamplesPerSec = dword;
            bytes_remaining -= 2;
            printf("Sample rate: %u\n", wInfo->dwSamplesPerSec);

            // Process the byte rate
            read_uint32(dword, file);
            cpu_uint32(dword, wInfo->endian);
            wInfo->dwAvgBytesPerSec = dword;
            bytes_remaining -= 4;
            printf("Byte rate: %u\n", wInfo->dwAvgBytesPerSec);

            // Process block alignment
            read_uint16(word, file);
            cpu_uint16(word, wInfo->endian);
            wInfo->wBlockAlign = word;
            bytes_remaining -= 4;
            printf("Block align: %u\n", wInfo->wBlockAlign);

            // Process bits per sample
            read_uint16(word, file);
            cpu_uint16(word, wInfo->endian);
            wInfo->wBitsPerSample = word;
            bytes_remaining -= 2;
            printf("Bits/sample: %u\n", wInfo->wBitsPerSample);

            // At this point, we expect there to be no more data for this
            // chunk if the audio format is 1 (PCM). The RIFF standard
            // requires that no additional fields be added if the audio
            // format is PCM. However, some encoders still add one or
            // two of these fields. Therefore, we skip the remaining
            // bytes of this chunk if our audio format is PCM.
            // Otherwise, we will record the field values.
            if(bytes_remaining == 0)
                 continue;
            else if(bytes_remaining > 0 && wInfo->wFormatTag == WAVE_FORMAT_PCM) {
                if(skip_bytes(bytes_remaining, file) == bytes_remaining)
                    printf("remaining %u bytes are being skipped for PCM.\n",
bytes_remaining);
                else {
                    printf("The file terminated prematurely, or could not allocate enough
memory.");
                    return -1;
                }
            }
            else if(bytes_remaining < 0) // Was the wrong chunk size given to us?
                return -1;
            else {
                // Process extra params
                read_uint16(word, file);
                cpu_uint16(word, wInfo->endian);
                wInfo->wExtraParamSize = word;
                bytes_remaining -= 2;
                printf("Extra param size: %04X\n", wInfo->wExtraParamSize);
```

```
                // allocate extra params based on extra param size
                wInfo->extraParams = (uint8_t *)malloc(wInfo->wExtraParamSize);

                if(wInfo->extraParams == NULL) {
                    printf("Could not allocate enough memory.\n");
                    return -1;
                }

                if(read_bytes(wInfo->extraParams, wInfo->wExtraParamSize, file) < wInfo-
>wExtraParamSize) {
                    printf("The file terminated prematurely or is corrupt.\n");
                    return -1;
                }
                bytes_remaining -= wInfo->wExtraParamSize;

                // Now we expect bytes remaining to be zero (0). If not, error
                if(bytes_remaining != 0) {
                    printf("The file could not be read or is corrupt.\n");
                    return -1;
                }
            }
            printf("\n");
            break;
        case DATA:
            printf("<data-ck>:\n");

            // Process the chunk size
            read_uint32(dword, file);
            cpu_uint32(dword, wInfo->endian);
            wInfo->data_chunk_size = dword;
            bytes_remaining = wInfo->data_chunk_size;
            printf("chunk size: %u\n", wInfo->data_chunk_size);

            // Process the WAVE data
            // allocate extra params based on extra param size
            wInfo->data = (uint8_t *)malloc(wInfo->data_chunk_size);
            if(read_bytes((uint8_t *)wInfo->data, wInfo->data_chunk_size, file) < wInfo-
>data_chunk_size) {
                printf("The file terminated prematurely or is corrupt.\n");
                return -1;
            }
            bytes_remaining -= wInfo->data_chunk_size;

            // Now we expect bytes remaining to be zero (0). If not, error
            if(bytes_remaining != 0) {
                printf("The file could not be read or is corrupt.\n");
                return -1;
            }
            break;
        }
    }
    fclose(file);
    printf("\nDone decoding WAVE file.\n\n");
    return 1;
}

int main(int argc, char *argv[])
```

```
{
    WaveInfo wInfo;
    if(decodeWave(argv[1], &wInfo) == -1) {
        printf("Failed to decode input file.\n");
        return 0;
    }
    playWave(&wInfo);
    return 0;
}
```

```
#include "globals.h"
#include <SDL/SDL_mixer.h>

void playWave(WaveInfo *wInfo)
{
    SDL_Surface *screen, *image, *image_loading, *temp;
    Mix_Chunk sound;        //Pointer to our sound, in memory
    int channel;                            //Channel on which our sound is played
    Uint16 audio_format; //Format of the audio we're playing
    int quit;
    SDL_Event event;

    int audio_rate_orignal = wInfo->dwSamplesPerSec; // Store original freq. for later
    int audio_rate = wInfo->dwSamplesPerSec; //Frequency of audio playback

    if(wInfo->wBitsPerSample == 8)
        audio_format = AUDIO_S8;
    else
        audio_format = AUDIO_S16SYS;

    int audio_channels = wInfo->wChannels;                    //2 channels = stereo
    int audio_buffers = wInfo->data_chunk_size;        //Size of the audio buffers in
memory

    //Initialize BOTH SDL video and SDL audio
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0) {
        printf("Unable to initialize SDL: %s\n", SDL_GetError());
        return;
    }

    //Initialize SDL_mixer with our chosen audio settings
    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels, audio_buffers) != 0) {
        printf("Unable to initialize audio: %s\n", Mix_GetError());
        exit(1);
    }

    sound.abuf = (uint8_t *)wInfo->data;
    sound.alen = wInfo->data_chunk_size;
    sound.allocated = 1;
    sound.volume = 128;

    //Set the video mode to anything, just need a window

    screen = SDL_SetVideoMode(531, 431, 16, SDL_DOUBLEBUF | SDL_ANYFORMAT);
    if (screen == NULL) {
        printf("Unable to set video mode: %s\n", SDL_GetError());
        return;
    }

    temp = SDL_LoadBMP("Einstein.bmp");
    if (temp == NULL) {
        printf("Unable to load bitmap: %s\n", SDL_GetError());
        return;
    }
    image = SDL_DisplayFormat(temp);

    temp = SDL_LoadBMP("Loading.bmp");
    if (temp == NULL) {
```

```
    printf("Unable to load bitmap: %s\n", SDL_GetError());
    return;
}
image_loading = SDL_DisplayFormat(temp);
SDL_FreeSurface(temp);

SDL_Rect src, dest;

src.x = 0;
src.y = 0;
src.w = image->w;
src.h = image->h;

dest.x = 0;
dest.y = 0;
dest.w = image->w;
dest.h = image->h;

SDL_BlitSurface(image, &src, screen, &dest);
SDL_Flip(screen);

//Play our sound file, and capture the channel on which it is played
channel = Mix_PlayChannel(-1, &sound, 0);
if(channel == -1) {
    printf("Unable to play WAV file: %s\n", Mix_GetError());
}
printf("Starting audio playback. See the pop-up window...\n");

/* Enable Unicode translation */
SDL_EnableUNICODE( 1 );
quit = 0;
/* Loop until an SDL_QUIT event is found */
while( !quit ){

    /* Poll for events */
    while(SDL_PollEvent(&event)){

        switch(event.type){
            /* Keyboard event */
            /* Pass the event data onto PrintKeyInfo() */
        case SDL_KEYUP:
            switch(event.key.keysym.sym) {
            case SDLK_ESCAPE: // Escape
                SDL_BlitSurface(image_loading, &src, screen, &dest);
                SDL_Flip(screen);
                printf("[Command - Esc] Stopping audio playback and exiting...\n");
                quit = 1;
                break;
            case SDLK_r: // Restart
                SDL_BlitSurface(image_loading, &src, screen, &dest);
                SDL_Flip(screen);
                printf("[Command - R] Restarting audio playback...\n");
                channel = Mix_PlayChannel(channel, &sound, 0);
                SDL_Delay(2000);
                SDL_BlitSurface(image, &src, screen, &dest);
                SDL_Flip(screen);
                break;
            case SDLK_s: // Stop
```

```
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - S] Pauing audio playback...\n");
                    Mix_Pause(channel);
                    SDL_Delay(2000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    break;
                case SDLK_p: // Play
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    if(Mix_Playing(channel) != 0) {
                        printf("[Command - P] Resuming audio playback from previous
point...\n");
                        Mix_Resume(channel);
                    } else {
                        printf("[Command - P] Resuming audio playback from
beginning...\n");
                        channel = Mix_PlayChannel(channel, &sound, 0);
                    }
                    SDL_Delay(2000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    break;
                case SDLK_1: // Original
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, audio_rate_orignal);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 1] Sample rate changed to %u Hz (Original)...\n",
wInfo->dwSamplesPerSec);
                    break;
                case SDLK_2: // 8kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 8000);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }
```

```
                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 2] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_3: // 11kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 16000);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 3] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_4: // 11kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 22050);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 4] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_5: // 32kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 32000);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
```

```
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 5] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_6: // 44kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 44056);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 6] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_7: // 88kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 88200);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 7] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_8: // 96kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
```

```
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 96000);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 8] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_9: // 176kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 176400);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
                    printf("[Command - 9] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                case SDLK_0: // 192kHz
                    SDL_BlitSurface(image_loading, &src, screen, &dest);
                    SDL_Flip(screen);
                    Mix_CloseAudio();
                    changeSampleRate(wInfo, 192000);

                    //Let's play back at new rate
                    audio_rate = wInfo->dwSamplesPerSec;
                    if(Mix_OpenAudio(audio_rate, audio_format, audio_channels,
audio_buffers) != 0) {
                        printf("Unable to re-initialize audio: %s\n", Mix_GetError());
                        exit(1);
                    }

                    channel = Mix_PlayChannel(channel, &sound, 0);
                    SDL_Delay(3000);
                    SDL_BlitSurface(image, &src, screen, &dest);
                    SDL_Flip(screen);
```

```
                    printf("[Command - 0] Sample rate changed to %u Hz...\n", wInfo-
>dwSamplesPerSec);
                    break;
                }
                break;
            /* SDL_QUIT event (window close) */
            case SDL_QUIT:
                SDL_BlitSurface(image_loading, &src, screen, &dest);
                SDL_Flip(screen);
                printf("[Command - quit] Stopping audio playback and exiting...\n");
                quit = 1;
                break;
            default:
                break;
            }
        }
    }

    SDL_FreeSurface(image);
    //Need to make sure that SDL_mixer and SDL have a chance to clean up
    Mix_CloseAudio();
    SDL_Quit();

    printf("\naudio playback complete\n\n");

    //Return success!
    return;
}
```

Appendix C: Links to references used

1. https://ccrma.stanford.edu/courses/422/projects/WaveFormat/
2. http://en.wikipedia.org/wiki/Pulse-code_modulation
3. http://www.mediacollege.com/audio/01/sound-waves.html
4. http://www.ascii-code.com/
5. http://www.libsdl.org/projects/SDL_mixer/
6. http://www.libsdl.org/download-1.2.php
7. http://www.daubnet.com/en/file-format-riff
8. http://www.digitalpreservation.gov/formats/fdd/fdd000025.shtml
9. http://sharkysoft.com/archive/lava/docs/javadocs/lava/riff/wave/doc-files/riffwave-content.htm
10. http://netghost.narod.ru/gff/vendspec/micriff/ms_riff.txt
11. http://www-mmsp.ece.mcgill.ca/documents/AudioFormats/WAVE/Samples.html
12. http://freewavesamples.com/
13. http://www.ibm.com/developerworks/aix/library/au-endianc/index.html?ca=drs-
14. http://www.ibm.com/developerworks/power/library/pa-spec16/?ca=dgr-lnxw07IFF
15. http://wiki.libsdl.org/moin.cgi/SDLKeycodeLookup