WAVE File Format

WAVE File Format is a file format for storing digital audio (waveform) data. It supports a variety of bit resolutions, sample rates, and channels of audio. This format is very popular upon IBM PC (clone) platforms, and is widely used in professional programs that process digital audio waveforms. It takes into account some pecularities of the Intel CPU such as little endian byte order.

This format uses Microsoft's version of the Electronic Arts Interchange File Format method for storing data in "chunks".

Data Types

A C-like language will be used to describe the data structures in the file. A few extra data types that are not part of standard C, but which will be used in this document, are:

pstring Pascal-style string, a one-byte count followed by that many text bytes. The total number of bytes in this data type should be even. A pad byte can be added to the end of the text to accomplish this. This pad byte is not reflected in the count.

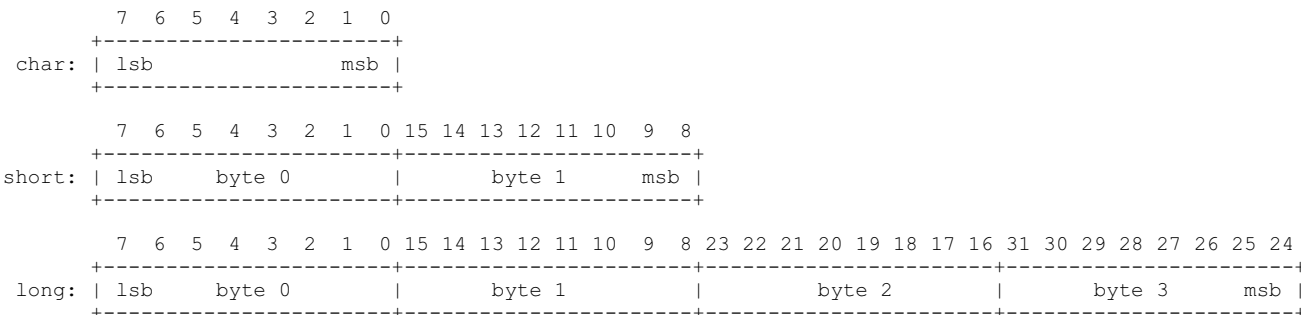**ID** A chunk ID (ie, 4 ASCII bytes).

Also note that when you see an array with no size specification (e.g., char ckData[];), this indicates a variable-sized array in our C-like language. This differs from standard C arrays.

Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

Data Organization

All data is stored in 8-bit bytes, arranged in Intel 80x86 (ie, little endian) format. The bytes of multiple-byte values are stored with the low-order (ie, least significant) bytes first. Data bits are as follows (ie, shown with bit numbers on top):

```
        7  6  5  4  3  2  1  0
      +----------------------+
char: | lsb              msb |
      +----------------------+

        7  6  5  4  3  2  1  0 15 14 13 12 11 10  9  8
      +----------------------+----------------------+
short:| lsb     byte 0       |        byte 1    msb |
      +----------------------+----------------------+

        7  6  5  4  3  2  1  0 15 14 13 12 11 10  9  8 23 22 21 20 19 18 17 16 31 30 29 28 27 26 25 24
      +----------------------+----------------------+----------------------+----------------------+
long: | lsb     byte 0       |        byte 1        |        byte 2        |      byte 3      msb |
      +----------------------+----------------------+----------------------+----------------------+
```
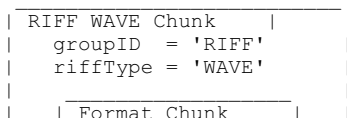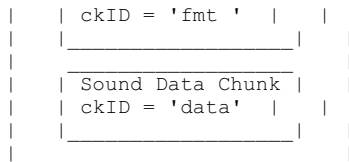
File Structure

A WAVE file is a collection of a number of different types of chunks. There is a required Format ("fmt ") chunk which contains important parameters describing the waveform, such as its sample rate. The Data chunk, which contains the actual waveform data, is also required. All other chunks are optional. Among the other optional chunks are ones which define cue points, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in the following sections of this document.

All applications that use WAVE must be able to read the 2 required chunks and can choose to selectively ignore the optional chunks. A program that copies a WAVE should copy all of the chunks in the WAVE, even those it chooses not to interpret.

There are no restrictions upon the order of the chunks within a WAVE file, with the exception that the Format chunk must precede the Data chunk. Some inflexibly written programs expect the Format chunk as the first chunk (after the RIFF header) although they shouldn't because the specification doesn't require this.

Here is a graphical overview of an example, minimal WAVE file. It consists of a single WAVE containing the 2 required chunks, a Format and a Data Chunk.

```
 _____
| RIFF WAVE Chunk      |
|   groupID  = 'RIFF'  |
|   riffType = 'WAVE'  |
|    _____    |
|   | Format Chunk |   |
```

```
|   |  ckID = 'fmt '  |    |
|   |_____|    |
|    _____     |
|   | Sound Data Chunk  |    |
|   |  ckID = 'data'    |    |
|   |_____|    |
|_____|
```

A Bastardized Standard

The WAVE format is sort of a bastardized standard that was concocted by too many "cooks" who didn't properly coordinate the addition of "ingredients" to the "soup". Unlike with the AIFF standard which was mostly designed by a small, coordinated group, the WAVE format has had all manner of much-too-independent, uncoordinated aberrations inflicted upon it. The net result is that there are far too many chunks that may be found in a WAVE file -- many of them duplicating the same information found in other chunks (but in an unnecessarily different way) simply because there have been too many programmers who took too many liberties with unilaterally adding their own additions to the WAVE format without properly coming to a concensus of what everyone else needed (and therefore it encouraged an "every man for himself" attitude toward adding things to this "standard"). One example is the Instrument chunk versus the Sampler chunk. Another example is the Note versus Label chunks in an Associated Data List. I don't even want to get into the totally irresponsible proliferation of compressed formats. (ie, It seems like everyone and his pet Dachshound has come up with some compressed version of storing wave data -- like we need 100 different ways to do that). Furthermore, there are lots of inconsistencies, for example how 8-bit data is unsigned, but 16-bit data is signed.

I've attempted to document only those aspects that you're very likely to encounter in a WAVE file. I suggest that you concentrate upon these and refuse to support the work of programmers who feel the need to deviate from a standard with inconsistent, proprietary, self-serving, unnecessary extensions. Please do your part to rein in half-ass programming.
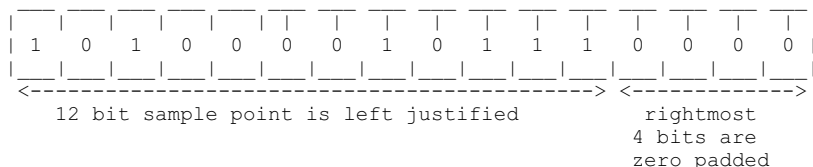
Sample Points and Sample Frames

A large part of interpreting WAVE files revolves around the two concepts of sample points and sample frames.
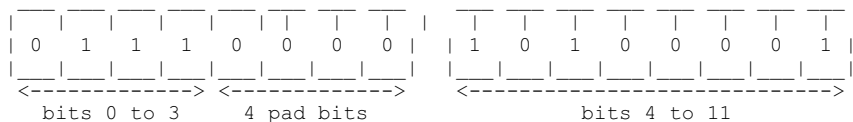
A sample point is a value representing a sample of a sound at a given moment in time. For waveforms with greater than 8-bit resolution, each sample point is stored as a linear, 2's-complement value which may be from 9 to 32 bits wide (as determined by the wBitsPerSample field in the Format Chunk, assuming PCM format -- an uncompressed format). For example, each sample point of a 16-bit waveform would be a 16-bit word (ie, two 8-bit bytes) where 32767 (0x7FFF) is the highest value and -32768 (0x8000) is the lowest value. For 8-bit (or less) waveforms, each sample point is a linear, unsigned byte where 255 is the highest value and 0 is the lowest value. Obviously, this signed/unsigned sample point discrepancy between 8-bit and larger resolution waveforms was one of those "oops" scenarios where some Microsoft employee decided to change the sign sometime after 8-bit wave files were common but 16-bit wave files hadn't yet appeared.

Because most CPU's read and write operations deal with 8-bit bytes, it was decided that a sample point should be rounded up to a size which is a multiple of 8 when stored in a WAVE. This makes the WAVE easier to read into memory. If your ADC produces a sample point from 1 to 8 bits wide, a sample point should be stored in a WAVE as an 8-bit byte (ie, unsigned char). If your ADC produces a sample point from 9 to 16 bits wide, a sample point should be stored in a WAVE as a 16-bit word (ie, signed short). If your ADC produces a sample point from 17 to 24 bits wide, a sample point should be stored in a WAVE as three bytes. If your ADC produces a sample point from 25 to 32 bits wide, a sample point should be stored in a WAVE as a 32-bit doubleword (ie, signed long). Etc.

Furthermore, the data bits should be left-justified, with any remaining (ie, pad) bits zeroed. For example, consider the case of a 12-bit sample point. It has 12 bits, so the sample point must be saved as a 16-bit word. Those 12 bits should be left-justified so that they become bits 4 to 15 inclusive, and bits 0 to 3 should be set to zero. Shown below is how a 12-bit sample point with a value of binary 101000010111 is formatted left-justified as a 16-bit word.
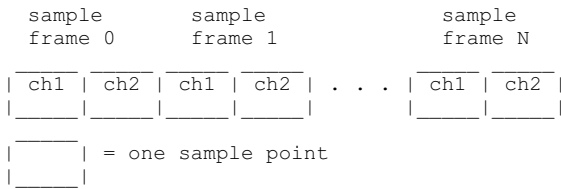
```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1   0   1   0   0   0   0   1   0   1   1   1   0   0   0   0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
  <------------------------------------------> <------------->
     12 bit sample point is left justified        rightmost
                                                   4 bits are
                                                   zero padded
```

But note that, because the WAVE format uses Intel little endian byte order, the LSB is stored first in the wave file as so:

```
 ___ ___ ___ ___ ___ ___ ___ ___   ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   | |   |   |   |   |   |   |   |   |
| 0   1   1   1   0   0   0   0 | | 1   0   1   0   0   0   0   1 |
|___|___|___|___|___|___|___|___| |___|___|___|___|___|___|___|___|
 <-------------> <------------->   <----------------------------->
  bits 0 to 3      4 pad bits                bits 4 to 11
```
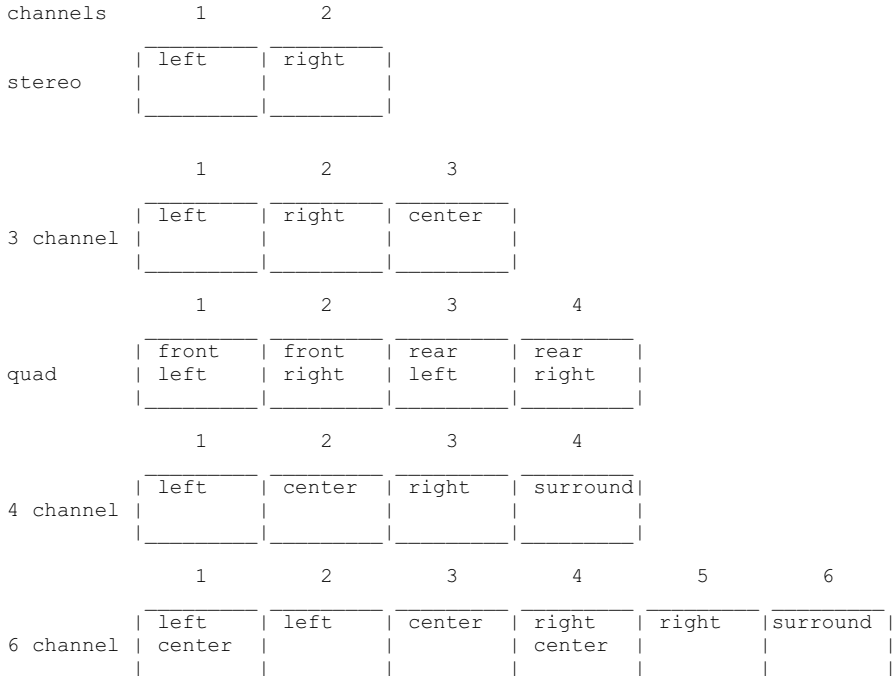
For multichannel sounds (for example, a stereo waveform), single sample points from each channel are interleaved. For example, assume a stereo (ie, 2 channel) waveform. Instead of storing all of the sample points for the left channel first, and then storing all of the sample points for the right channel next, you "mix" the two channels' sample points together. You would store the first sample point of the left channel. Next, you would store the first sample point of the right channel. Next, you would store the second sample point of the left channel. Next, you would store the second sample point of the right channel, and so on, alternating between storing the next sample point of each channel. This is what is meant by interleaved data; you store the next sample point of each of the channels in turn, so that the sample points that are meant to be "played" (ie, sent to a DAC) simultaneously are stored

contiguously.

The sample points that are meant to be "played" (ie, sent to a DAC) simultaneously are collectively called a **sample frame**. In the example of our stereo waveform, every two sample points makes up another sample frame. This is illustrated below for that stereo example.

```
   sample         sample              sample
   frame 0        frame 1             frame N
  _____       _____            _____
| ch1 | ch2 | ch1 | ch2 | . . . | ch1 | ch2 |
|_____|_____|_____|_____|       |_____|_____|
 _____
|     | = one sample point
|_____|
```

For a monophonic waveform, a sample frame is merely a single sample point (ie, there's nothing to interleave). For multichannel waveforms, you should follow the conventions shown below for which order to store channels within the sample frame. (ie, Below, a single sample frame is displayed for each example of a multichannel waveform).

```
   channels       1         2
                _____  _____
               | left    | right   |
   stereo      |         |         |
               |_____|_____|


                 1         2         3
                _____  _____  _____
               | left    | right   | center  |
   3 channel   |         |         |         |
               |_____|_____|_____|

                 1         2         3         4
                _____  _____  _____  _____
               | front   | front   | rear    | rear    |
   quad        | left    | right   | left    | right   |
               |_____|_____|_____|_____|

                 1         2         3         4
                _____  _____  _____  _____
               | left    | center  | right   | surround|
   4 channel   |         |         |         |         |
               |_____|_____|_____|_____|

                 1         2         3         4         5         6
                _____  _____  _____  _____  _____  _____
               | left    | left    | center  | right   | right   |surround |
   6 channel   | center  |         |         | center  |         |         |
               |_____|_____|_____|_____|_____|_____|
```

The sample points within a sample frame are packed together; there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

Note that the above discussion outlines the format of data within an uncompressed data chunk. There are some techniques of storing compressed data in a data chunk. Obviously, that data would need to be uncompressed, and then it will adhere to the above layout.

The Format Chunk

The Format (fmt) chunk describes fundamental parameters of the waveform data such as sample rate, bit resolution, and how many channels of digital audio are stored in the WAVE.

```
#define FormatID 'fmt '   /* chunkID for Format Chunk. NOTE: There is a space at the end of this ID. */

typedef struct {
  ID             chunkID;
  long           chunkSize;

  short          wFormatTag;
  unsigned short wChannels;
  unsigned long  dwSamplesPerSec;
  unsigned long  dwAvgBytesPerSec;
  unsigned short wBlockAlign;
  unsigned short wBitsPerSample;

/* Note: there may be additional fields here, depending upon wFormatTag. */
```

```
} FormatChunk;
```

The ID is always "**fmt** ". The chunkSize field is the number of bytes in the chunk. This does not include the 8 bytes used by ID and Size fields. For the Format Chunk, chunkSize may vary according to what "format" of WAVE file is specified (ie, depends upon the value of wFormatTag).

WAVE data may be stored without compression, in which case the sample points are stored as described in **Sample Points and Sample Frames**. Alternately, different forms of compression may be used when storing the sound data in the Data chunk. With compression, each sample point may take a differing number of bytes to store. The wFormatTag indicates whether compression is used when storing the data.

If compression is used (ie, WFormatTag is some value other than 1), then there will be additional fields appended to the Format chunk which give needed information for a program wishing to retrieve and decompress that stored data. The first such additional field will be an unsigned short that indicates how many more bytes have been appended (after this unsigned short). Furthermore, compressed formats must have a Fact chunk which contains an unsigned long indicating the size (in sample points) of the waveform after it has been decompressed. There are (too) many compressed formats. Details about them can be gotten from Microsoft's web site.

If no compression is used (ie, wFormatTag = 1), then there are no further fields.

The wChannels field contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, 4 means four channel sound, etc. Any number of audio channels may be represented. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame.

The actual waveform data is stored in another chunk, the Data Chunk, which will be described later.

The dwSamplesPerSec field is the sample rate at which the sound is to be played back in sample frames per second (ie, Hertz). The 3 standard MPC rates are 11025, 22050, and 44100 KHz, although other rates may be used.

The dwAvgBytesPerSec field indicates how many bytes play every second. dwAvgBytesPerSec may be used by an application to estimate what size RAM buffer is needed to properly playback the WAVE without latency problems. Its value should be equal to the following formula rounded up to the next whole number:

dwSamplesPerSec * wBlockAlign

The wBlockAlign field should be equal to the following formula, rounded to the next whole number:

wChannels * (wBitsPerSample % 8)

Essentially, wBlockAlign is the size of a sample frame, in terms of bytes. (eg, A sample frame for a 16-bit mono wave is 2 bytes. A sample frame for a 16-bit stereo wave is 4 bytes. Etc).

The wBitsPerSample field indicates the bit resolution of a sample point (ie, a 16-bit waveform would have wBitsPerSample = 16).

One, and only one, Format Chunk is required in every WAVE.

Data Chunk

The Data (data) chunk contains the actual sample frames (ie, all channels of waveform data).

```
#define DataID 'data'  /* chunk ID for data Chunk */

typedef struct {
  ID            chunkID;
  long          chunkSize;

  unsigned char waveformData[];
} DataChunk;
```

The ID is always **data**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

Remember that the bit resolution, and other information is gotten from the Format chunk.

The following discussion assumes uncompressed data.

The waveformData array contains the actual waveform data. The data is arranged into what are called *sample frames*. For more information on the arrangement of data, see "Sample Points and Sample Frames".

You can determine how many bytes of actual waveform data there is from the Data chunk's chunkSize field. The number of sample frames in waveformData is determined by dividing this chunkSize by the Format chunk's wBlockAlign.

The Data Chunk is required. One, and only one, Data Chunk may appear in a WAVE.

Another way of storing waveform data

So, you're thinking "This WAVE format isn't that bad. It seems to make sense and there aren't all that many inconsistencies, duplications, and inefficiencies". You fool! We're just getting started with our first excursion into unnecessary inconsistencies, duplications, and inefficiency.

Sure, countless brain-damaged programmers have inflicted literally dozens of compressed data formats upon the Data chunk, but apparently someone felt that even this wasn't enough to make your life difficult in trying to support WAVE files. No, some half-wit decided that it would be a good idea to screw around with storing waveform data in something other than one Data chunk. NOOOOOOOOOOOOOO!!!!!!

For some god-forsaken reason, someone came up with the idea of using an imbedded IFF List inside of the WAVE file. NOOOOOOOOOOOOOOOOOO!!!!!!!! And this "Wave List" would contain multiple 'data' and 'slnt' chunks. NOOOOOOOOOOOOOOOOO!!!! The Type ID for this List is 'wavl'.

I strongly suggest that you refuse to support any WAVE file that exhibits this Wave List nonsense. There's no need for it, and hopefully, the misguided programmer who conjured it up will be embarrassed into hanging his head in shame when nobody agrees to support his foolishness. Just say "NOOOOOOOOOOOOOOO!!!!"

Cue Chunk

The Cue chunk contains one or more "cue points" or "markers". Each cue point references a specific offset within the waveformData array, and has its own CuePoint structure within this chunk.

In conjunction with the Playlist chunk, the Cue chunk can be used to store looping information.

CuePoint Structure

```
typedef struct {
  long    dwIdentifier;
  long    dwPosition;
  ID      fccChunk;
  long    dwChunkStart;
  long    dwBlockStart;
  long    dwSampleOffset;
} CuePoint;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other CuePoint structure). This is used to associate a CuePoint structure with other structures used in other chunks which will be described later.

The dwPosition field specifies the position of the cue point within the "play order" (as determined by the Playlist chunk. See that chunk for a discussion of the play order).

The fccChunk field specifies the chunk ID of the Data or Wave List chunk which actually contains the waveform data to which this CuePoint refers. If there is only one Data chunk in the file, then this field is set to the ID 'data'. On the other hand, if the file contains a Wave List (which can contain both 'data' and 'slnt' chunks), then fccChunk will specify 'data' or 'slnt' depending upon in which type of chunk the referenced waveform data is found.

The dwChunkStart and dwBlockStart fields are set to 0 for an uncompressed WAVE file that contains one 'data' chunk. These fields are used only for WAVE files that contain a Wave List (with multiple 'data' and 'slnt' chunks), or for a compressed file containing a 'data' chunk. (Actually, in the latter case, dwChunkStart is also set to 0, and only dwBlockStart is used). Again, I want to emphasize that you can avoid all of this unnecessary crap if you avoid hassling with compressed files, or Wave Lists, and instead stick to the sensible basics.

The dwChunkStart field specifies the byte offset of the start of the 'data' or 'slnt' chunk which actually contains the waveform data to which this CuePoint refers. This offset is relative to the start of the first chunk within the Wave List. (ie, It's the byte offset, within the Wave List, of where the 'data' or 'slnt' chunk of interest appears. The first chunk within the List would be at an offset of 0).

The dwBlockStart field specifies the byte offset of the start of the block containing the position. This offset is relative to the start of the waveform data within the 'data' or 'slnt' chunk.

The dwSampleOffset field specifies the sample offset of the cue point relative to the start of the block. In an uncompressed file, this equates to simply being the offset within the waveformData array. Unfortunately, the WAVE documentation is much too ambiguous, and doesn't define what it means by the term "sample offset". This could mean a byte offset, or it could mean counting the sample points (for example, in a 16-bit wave, every 2 bytes would be 1 sample point), or it could even mean sample frames (as the loop offsets in AIFF are specified). Who knows? The guy who conjured up the Cue chunk certainly isn't saying. I'm assuming that it's a byte offset, like the above 2 fields.

Cue Chunk

```
#define CueID 'cue '  /* chunk ID for Cue Chunk */

typedef struct {
  ID        chunkID;
  long      chunkSize;

  long      dwCuePoints;
  CuePoint  points[];
} CueChunk;
```

The ID is always **cue** . chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields.

The dwCuePoints field is the number of CuePoint structures in the Cue Chunk. If dwCuePoints is not 0, it is followed by that many CuePoint structures, one after the other. Because all fields in a CuePoint structure are an even number of bytes, the length of any CuePoint will always be even. Thus, CuePoints are packed together with no unused bytes between them. The CuePoints need not be placed in any particular order.

The Cue chunk is optional. No more than one Cue chunk can appear in a WAVE.

Playlist chunk

The Playlist (plst) chunk specifies a play order for a series of cue points. The Cue chunk contains all of the cue points, but the Playlist chunk determines how those cue points are used when playing back the waveform (ie, which cue points represent looped sections, and in what order those loops are "played"). The Playlist chunk contains one or more Segment structures, each of which identifies a looped section of the waveform (in conjunction with the CuePoint structure with which it is associated).

Segment Structure

```
typedef struct {
  long    dwIdentifier;
  long    dwLength;
  long    dwRepeats;
} Segment;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other Segment structure). This field should correspond with the dwIndentifier field of some CuePoint stored in the Cue chunk. In other words, this Segment structure contains the looping information associated with that CuePoint structure with the same ID number.

The dwLength field specifies the length of the section in samples (ie, the length of the looped section). Note that the start position of the loop would be the dwSampleOffset of the referenced CuePoint structure in the Cue chunk. (Or, you may need to hassle with the dwChunkStart and dwBlockStart fields as well if dealing with a Wave List or compressed data).

The dwRepeats field specifies the number of times to play the loop. I assume that a value of 1 means to repeat this loop once only, but the WAVE documentation is very incomplete and omits this important information. I have no idea how you would specify an infinitely repeating loop. Certainly, the person who conjured up the Playlist chunk appears to have no idea whatsoever. Due to the ambiguities, inconsistencies, inefficiencies, and omissions of the Cue and Playlist chunks, I very much recommend that you use the Sampler chunk (described later) to replace them.

Playlist chunk

```
#define PlaylistID 'plst'  /* chunk ID for Playlist Chunk */

typedef struct {
  ID        chunkID;
  long      chunkSize;

  long      dwSegments;
  Segment   Segments[];
} PlaylistChunk;
```

The ID is always **plst**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields.

The dwSegments field is the number of Segment structures in the Playlist Chunk. If dwSegments is not 0, it is followed by that many Segment structures, one after the other. Because all fields in a Segment structure are an even number of bytes, the length of any Segment will always be even. Thus, Segments are packed together with no unused bytes between them. The Segments need not be placed in any particular order.

Associated Data List

The Associated Data List contains text "labels" or "names" that are associated with the CuePoint structures in the Cue chunk. In other words, this list contains the text labels for those CuePoints.

Again, we're talking about another imbedded IFF List within the WAVE file. NOOOOOOOOOOOOOO!!!! What's a List? A List is simply a "master chunk" that contains several "sub-chunks". Just like with any other chunk, the "master chunk" has an ID and chunkSize, but inside of this chunk are sub-chunks, each with its own ID and chunkSize. Of course, the chunkSize for the master chunk (ie, List) includes the size of all of these sub-chunks (including their ID and chunkSize fields).

The "Type ID" for the Associated Data List is "adtl". Remember that an IFF list header has 3 fields:

```
typedef struct {
  ID      listID;       /* 'list' */
  long    chunkSize;    /* includes the Type ID below */
  ID      typeID;       /* 'adtl' */
} ListHeader;
```

There are several sub-chunks that may be found inside of the Associated Data List. The ones that are important to WAVE format have IDs of "labl", "note", or "ltxt". Ignore the rest. Here are those 3 sub-chunks and their fields:

The Associated Data List is optional. The WAVE documentation doesn't specify if more than one can be contained in a WAVE file.

Label Chunk

```
#define LabelID 'labl'  /* chunk ID for Label Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  char    dwText[];
} LabelChunk;
```

The ID is always **labl**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwIdentifier field contains a unique number (ie, different than the ID number of any other Label chunk). This field should correspond with the dwIndentifier field of some CuePoint stored in the Cue chunk. In other words, this Label chunk contains the text label associated with that CuePoint structure with the same ID number.

The dwText array contains the text label. It should be a null-terminated string. (The null byte is included in the chunkSize, therefore the length of the string, including the null byte, is chunkSize - 4).

Note Chunk

```
#define NoteID 'note'  /* chunk ID for Note Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  char    dwText[];
} NoteChunk;
```

The Note chunk, whose ID is **note**, is otherwise exactly the same as the Label chunk (ie, same fields). See what I mean about pointless duplication? But, in theory, a Note chunk contains a "comment" about a CuePoint, whereas the Label chunk is supposed to contain the actual CuePoint label. So, it's possible that you'll find both a Note and Label for a specific CuePoint, each containing different text.

Labeled Text Chunk

```
#define LabelTextID 'ltxt'  /* chunk ID for Labeled Text Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  long    dwSampleLength;
  long    dwPurpose;
  short   wCountry;
  short   wLanguage;
```

```
    short    wDialect;
    short    wCodePage;
    char     dwText[];
} LabelTextChunk;
```

The ID is always **ltxt**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwIdentifier field is the same as the Label chunk.

The dwSampleLength field specifies the number of sample points in the segment of waveform data. In other words, a Labeled Text chunk contains a label for a **section** of the waveform data, not just a specific point, for example the looped section of a waveform.

The dwPurpose field specifies the type or purpose of the text. For example, dwPurpose can contain an ID like "scrp" for script text or "capt" for close-caption text. How is this related to waveform data? Well, it isn't really. It's just that Associated Data Lists are used in other file formats, so they contain generic fields that sometimes don't have much relevance to waveform data.

The wCountry, wLanguage, and wCodePage fields specify the country code, language/dialect, and code page for the text. An application typically queries these values from the operating system.

Sampler Chunk

The Sampler (smpl) Chunk defines basic parameters that an instrument, such as a MIDI sampler, could use to play the waveform data. Most importantly, it includes information about looping the waveform (ie, during playback, to "sustain" the waveform). Of course, as you've come to expect from the WAVE file format, it duplicates some of the information that can be found in the Cue and Playlist chunks, but fortunately, in a more sensible, consistent, better-documented way.

```
#define SamplerID 'smpl'  /* chunk ID for Sampler Chunk */

typedef struct {
    ID             chunkID;
    long           chunkSize;

    long           dwManufacturer;
    long           dwProduct;
    long           dwSamplePeriod;
    long           dwMIDIUnityNote;
    long           dwMIDIPitchFraction;
    long           dwSMPTEFormat;
    long           dwSMPTEOffset;
    long           cSampleLoops;
    long           cbSamplerData;
    struct SampleLoop Loops[];
} SamplerChunk;
```

The ID is always **smpl**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwManufacturer field contains the MMA Manufacturer code for the intended sampler. Each manufacturer of MIDI products has his own ID assigned to him by the MIDI Manufacturer's Association. See the MIDI Specification (under System Exclusive) for a listing of current Manufacturer IDs. The high byte of dwManufacturer indicates the number of low order bytes (1 or 3) that are valid for the manufacturer code. For example, this value will be 0x01000013 for Digidesign (the MMA Manufacturer code is one byte, 0x13); whereas 0x03000041 identifies Microsoft (the MMA Manufacturer code is three bytes, 0x00 0x00 0x41). If the WAVE is not intended for a specific manufacturer, then this field should be set to 0.

The dwProduct field contains the Product code (ie, model ID) of the intended sampler for the dwManufacturer. Contact the manufacturer of the sampler to ascertain the sampler's model ID. If the WAVE is not intended for a specific manufacturer's product, then this field should be set to 0.

The dwSamplePeriod field specifies the period of one sample in nanoseconds (normally 1/nSamplesPerSec from the Format chunk. But note that this field allows finer tuning than nSamplesPerSec). For example, 44.1 KHz would be specified as 22675 (0x00005893).

The dwMIDIUnityNote field is the MIDI note number at which the instrument plays back the waveform data without pitch modification (ie, at the same sample rate that was used when the waveform was created). This value ranges 0 through 127, inclusive. Middle C is 60.

The dwMIDIPitchFraction field specifies the fraction of a semitone up from the specified dwMIDIUnityNote. A value of 0x80000000 is 1/2 semitone (50 cents); a value of 0x00000000 represents no fine tuning between semitones.

The dwSMPTEFormat field specifies the SMPTE time format used in the dwSMPTEOffset field. Possible values are:

```
0  = no SMPTE offset (dwSMPTEOffset should also be 0)
24 = 24 frames per second
25 = 25 frames per second
29 = 30 frames per second with frame dropping ('30 drop')
30 = 30 frames per second
```

The dwSMPTEOffset field specifies a time offset for the sample if it is to be syncronized or calibrated according to a start time other than 0. The format of this value is 0xhhmmssff. hh is a signed Hours value [-23..23]. mm is an unsigned Minutes value [0..59]. ss is unsigned Seconds value [0..59]. ff is an unsigned value [0..(- 1)].

The cSampleLoops field is the number (count) of SampleLoop structures that are appended to this chunk. These structures immediately follow the cbSamplerData field. This field will be 0 if there are no SampleLoop structures.

The cbSamplerData field specifies the size (in bytes) of any optional fields that an application wishes to append to this chunk. An application which needed to save additional information (ie, beyond the above fields) may append additional fields to the end of this chunk, after all of the SampleLoop structures. These additional fields are also reflected in the ChunkSize, and remember that the chunk should be padded out to an even number of bytes. The cbSamplerData field will be 0 if no additional information is appended to the chunk.

What follows the above fields are any SampleLoop structures. Each SampleLoop structure defines one loop (ie, the start and end points of the loop, and how many times it plays). What follows any SampleLoop structures are any additional, proprietary sampler information that an application chooses to store.

SampleLoop Structure

```
typedef struct {
  long  dwIdentifier;
  long  dwType;
  long  dwStart;
  long  dwEnd;
  long  dwFraction;
  long  dwPlayCount;
} SampleLoop;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other SampleLoop structure). This field may correspond with the dwIdentifier field of some CuePoint stored in the Cue chunk. In other words, the CuePoint structure which has the same ID number would be considered to be describing the same loop as this SampleLoop structure. Furthermore, this field corresponds to the dwIndentifier field of any label stored in the Associated Data List. In other words, the text string (within some chunk in the Associated Data List) which has the same ID number would be considered to be this loop's "name" or "label".

The dwType field is the loop type (ie, how the loop plays back) as so:

```
0 - Loop forward (normal)
1 - Alternating loop (forward/backward)
2 - Loop backward
3-31 - reserved for future standard types
32-? - sampler specific types (manufacturer defined)
```

The dwStart field specifies the startpoint of the loop. In other words, it's the byte offset from the start of waveformData[], where an offset of 0 would be at the start of the waveformData[] array (ie, the loop start is at the very first sample point).

The dwEnd field specifies the endpoint of the loop (ie, a byte offset).

The dwFraction field allows fine-tuning for loop fractional areas between samples. Values range from 0x00000000 to 0xFFFFFFFF. A value of 0x80000000 represents 1/2 of a sample length.

The dwPlayCount field is the number of times to play the loop. A value of 0 specifies an infinite sustain loop (ie, the wave keeps looping until some external force interrupts playback, such as the musician releasing the key that triggered that wave's playback).

The Sampler Chunk is optional. I don't know as if there is any limit of one per WAVE file. I don't see why there should be such a limit, since after all, an application may need to deal with several MIDI samplers.

The Instrument Chunk Format

The Instrument Chunk contains some of the same type of information as the Sampler chunk. So what else is new?

```
#define InstrumentID 'inst'  /* chunkID for Instruments Chunk */

typedef struct {
  ID     chunkID;
  long   chunkSize;

  unsigned char UnshiftedNote;
```

```
    char        FineTune;
    char        Gain;
    unsigned char LowNote;
    unsigned char HighNote;
    unsigned char LowVelocity;
    unsigned char HighVelocity;
} InstrumentChunk;
```

The ID is always **inst**. chunkSize should always be 7 since there are no fields of variable length.

The UnshiftedNote field is the same as the Sampler chunk's dwMIDIUnityNote field.

The FineTune field determines how much the instrument should alter the pitch of the sound when it is played back. Units are in cents (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised. While not the same measurement is used, this field serves the same purpose as the Sampler chunk's dwFraction field.

The Gain field is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0db means no change, 6db means double the value of each sample point (ie, every additional 6db doubles the gain), while -6db means halve the value of each sample point.

The LowNote and HighNote fields specify the suggested MIDI note range on a keyboard for playback of the waveform data. The waveform data should be played if the instrument is requested to play a note between the low and high note numbers, inclusive. The UnshiftedNote does not have to be within this range.

The LowVelocity and HighVelocity fields specify the suggested range of MIDI velocities for playback of the waveform data. The waveform data should be played if the note-on velocity is between low and high velocity, inclusive. The range is 1 (lowest velocity) through 127 (highest velocity), inclusive.

The Instrument Chunk is optional. No more than 1 Instrument Chunk can appear in one WAVE.

Audio Interchange File Format (AIFF)

Audio Interchange File Format (or AIFF) is a file format for storing digital audio (waveform) data. It supports a variety of bit resolutions, sample rates, and channels of audio. This format is very popular upon Apple platforms, and is widely used in professional programs that process digital audio waveforms.

This format uses the Electronic Arts Interchange File Format method for storing data in "chunks". You should read the article About Interchange File Format before proceeding.

Data Types

A C-like language will be used to describe the data structures in the file. A few extra data types that are not part of standard C, but which will be used in this document, are:

extended 80 bit IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type Extended). This would be a 10 byte field.

pstring Pascal-style string, a one-byte count followed by that many text bytes. The total number of bytes in this data type should be even. A pad byte can be added to the end of the text to accomplish this. This pad byte is not reflected in the count.

ID A chunk ID (ie, 4 ASCII bytes) as described in About Interchange File Format.

Also note that when you see an array with no size specification (e.g., char ckData[];), this indicates a variable-sized array in our C-like language. This differs from standard C arrays.

Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

Data Organization

All data is stored in Motorola 68000 (ie, big endian) format. The bytes of multiple-byte values are stored with the high-order (ie, most significant) bytes first. Data bits are as follows (ie, shown with bit numbers on top):

```
       7  6  5  4  3  2  1  0
      +-----------------------+
char: | msb              lsb |
      +-----------------------+

      15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```