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).

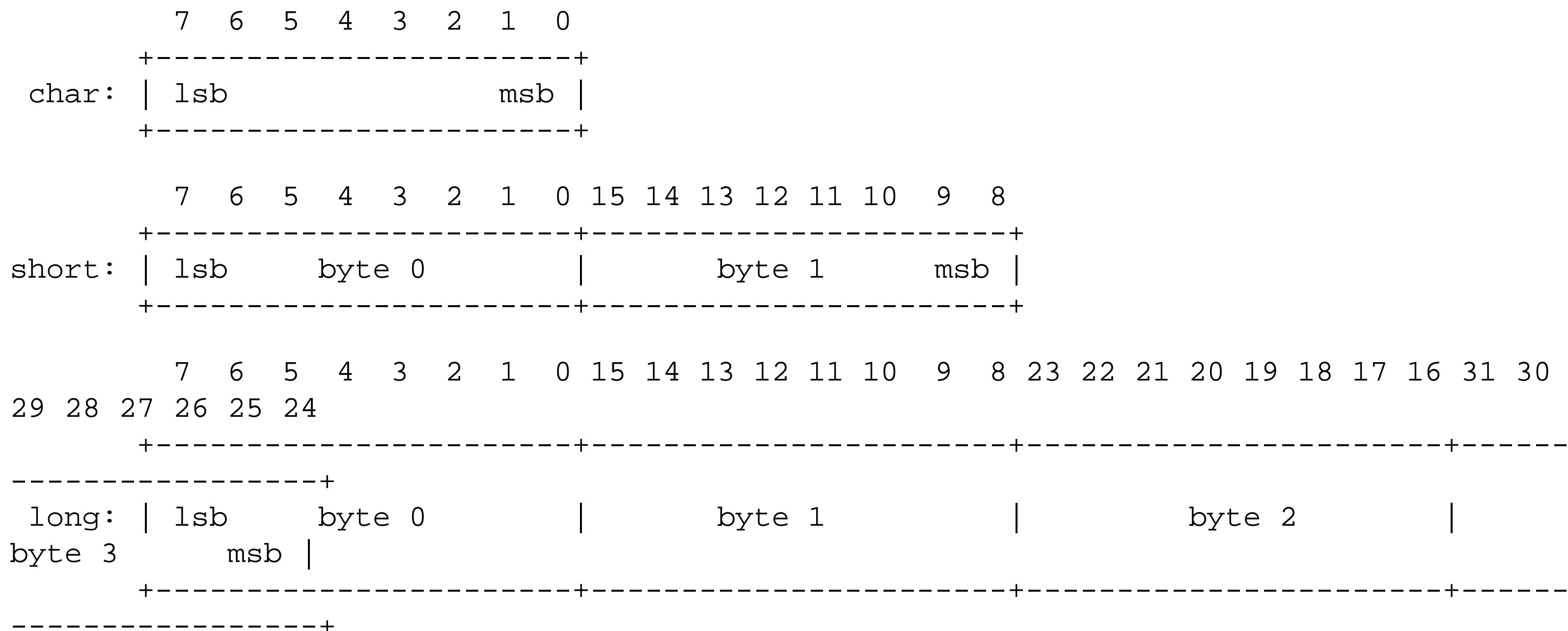
also note that when you see an array with no size specification (e.g., char ckdata[];), this indicates a variable-sized array in our c-like language. this differs from standard c arrays.

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):



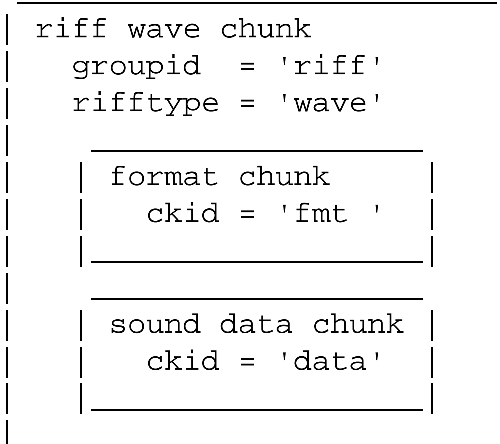
file structure

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.



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.

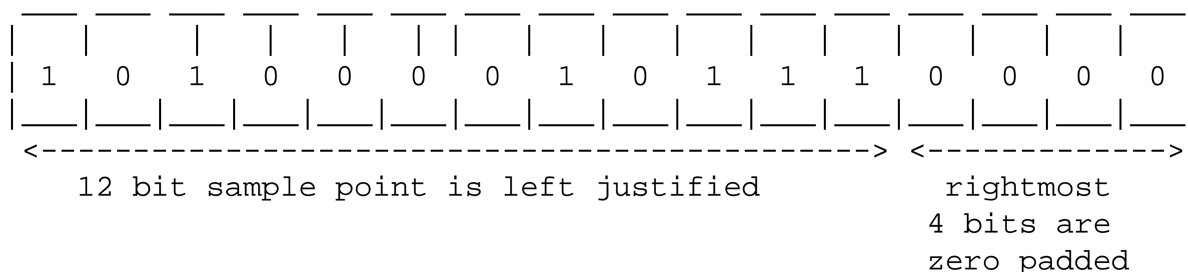
sample points and sample frames

a large part of interpreting wave files revolves around the two concepts of sample points and sample frames.

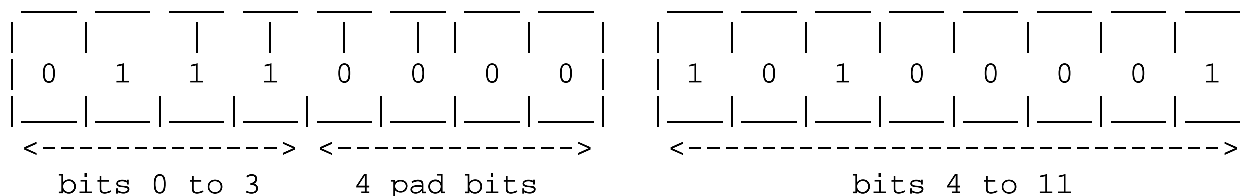
a sample point is a value representing a sample of a sound at a given moment in time. for waveforms with greater than 8-bit resolution, each sample point is stored as a linear, 2's-complement value which may be from 9 to 32 bits wide (as determined by the wbitspersample field in the format chunk, assuming 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.

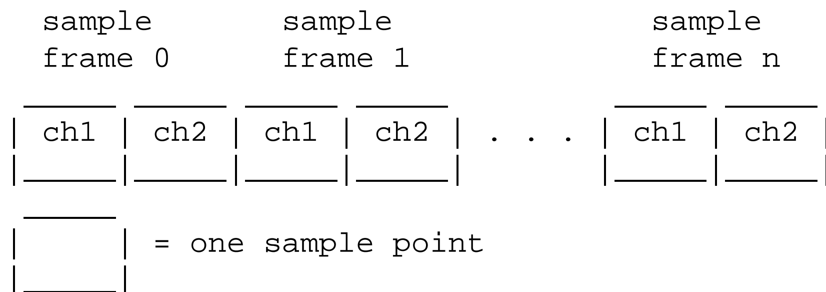


but note that, because the wave format uses intel little endian byte order, the lsb is stored first in the wave file as so:



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.



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 arrangment 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. noooooooooooooooo!!!! 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. nooooooooooooooo!!!! 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. 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*.

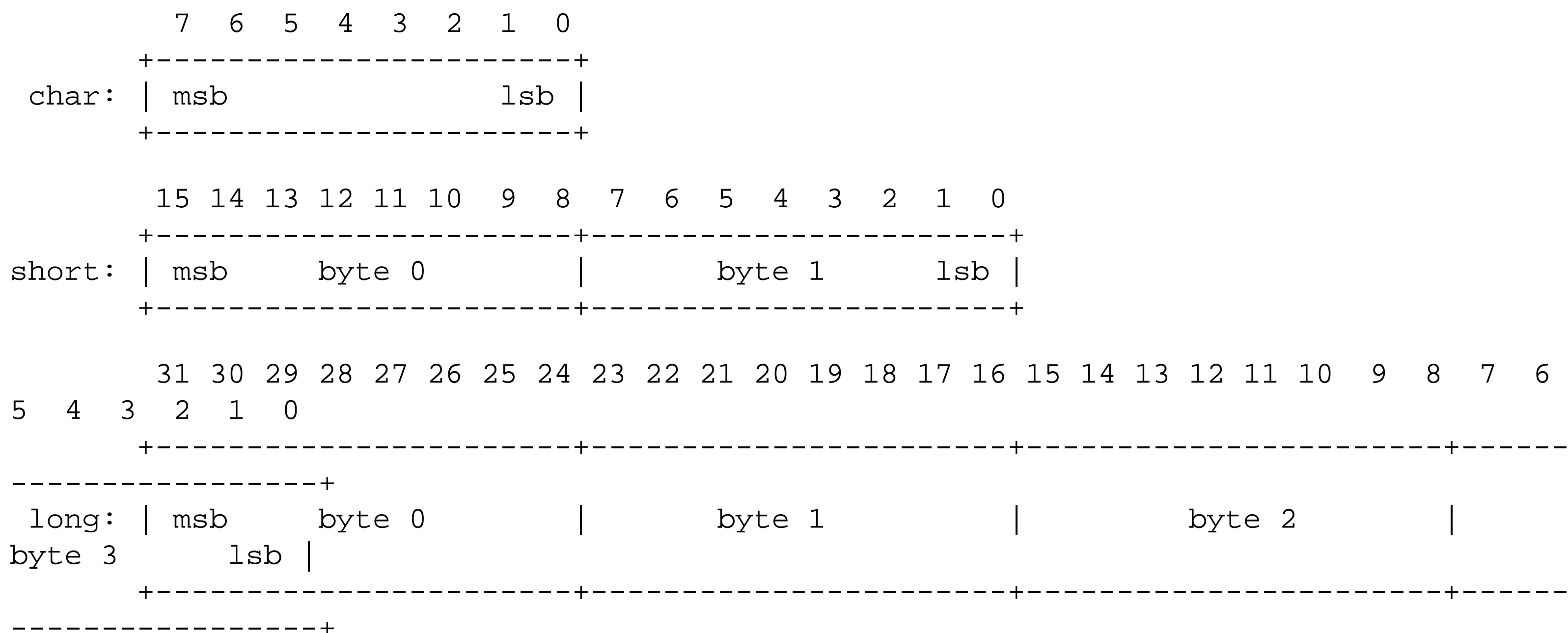
also note that when you see an array with no size specification (e.g., char ckdata[];), this indicates a variable-sized array in our c-like language. this differs from standard c arrays.

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
       +-----------------------+-----------------------+
short: | msb     byte 0        |       byte 1     lsb |
       +-----------------------+-----------------------+

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

file structure

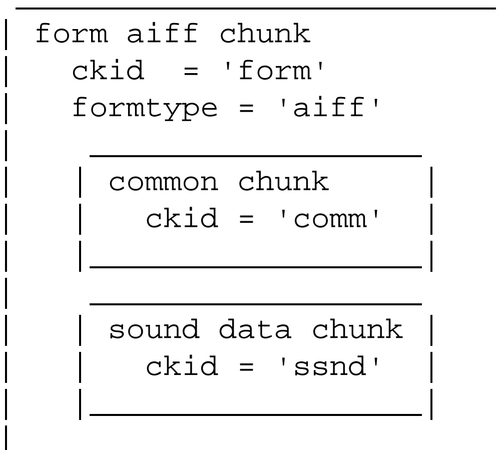
an audio iff file is a collection of a number of different types of chunks. there is a required common chunk which contains important parameters describing the waveform, such as its length and sample rate. the sound data chunk, which contains the actual waveform data, is also required if the waveform data has a length greater than 0 (ie, there actually is waveform data in the

form). all other chunks are optional. among the other optional chunks are ones which define markers, 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 form aiff must be able to read the 2 required chunks and can choose to selectively ignore the optional chunks. a program that copies a form aiff should copy all of the chunks in the form aiff, even those it chooses not to interpret.

there are no restrictions upon the order of the chunks within a form aiff.

here is a graphical overview of an example, minimal aiff file. it consists of a single form aiff containing the 2 required chunks, a common chunk and a sound data chunk.



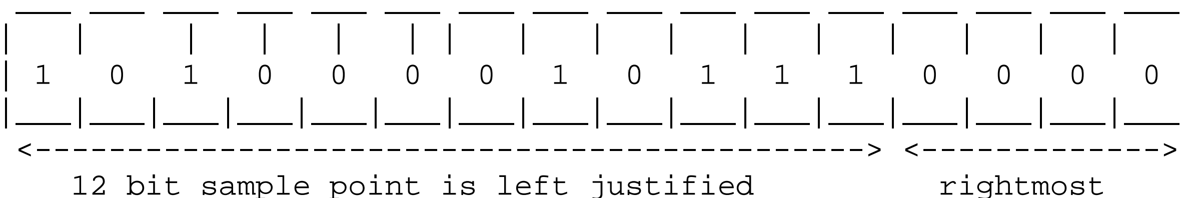
sample points and sample frames

a large part of interpreting audio iff 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. each sample point is stored as a linear, 2's-complement value which may be from 1 to 32 bits wide (as determined by the samplesize field in the common chunk). for example, each sample point of an 8-bit waveform would be an 8-bit byte (ie, a signed char).

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 an aiff. this makes the aiff 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 an aiff as an 8-bit byte (ie, signed char). if your adc produces a sample point from 9 to 16 bits wide, a sample point should be stored in an aiff 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 an aiff as three bytes. if your adc produces a sample point from 25 to 32 bits wide, a sample point should be stored in an aiff as a 32-bit doubleword (ie, signed long). etc.

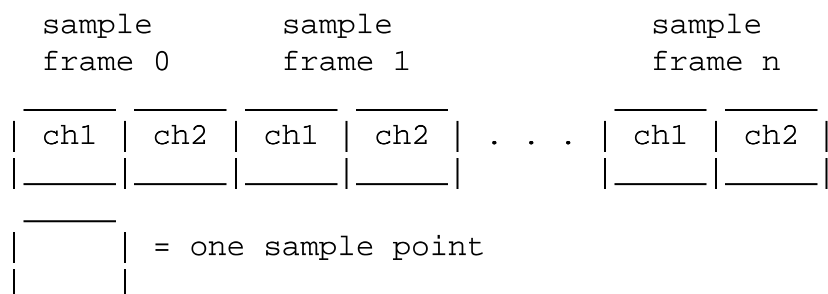
furthermore, the data bits should be left-justified, with any remaining (ie, pad) bits zeroed. for example, consider the case of a 12-bit sample point. it has 12 bits, so the sample point must be saved as a 16-bit word. those 12 bits should be left-justified so that they become bits 4 to 15 inclusive, 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 stored left-justified as two bytes (ie, a 16-bit word).



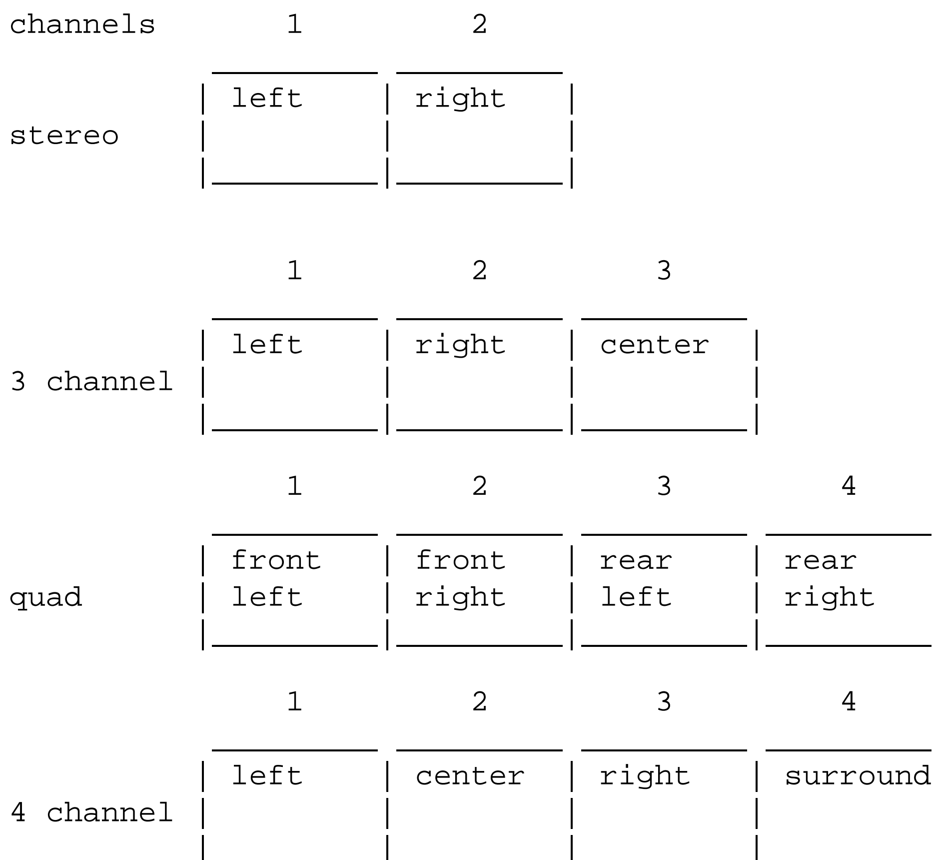
4 bits are
zero padded

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.



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).



```
                  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.

---

the common chunk

the common 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 form aiff.

```
#define commonid 'comm'   /* chunkid for common chunk */

typedef struct {
  id              chunkid;
  long            chunksize;

  short           numchannels;
  unsigned long   numsampleframes;
  short           samplesize;
  extended        samplerate;
} commonchunk;
```

the id is always **comm**. 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 common chunk, chunksize should always 18 since there are no fields of variable length (but to maintain compatibility with possible future extensions, if the chunksize is > 18, you should always treat those extra bytes as pad bytes).

the numchannels 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 sound data chunk, which will be described later.

the numsampleframes field contains the number of sample frames. this is not necessarily the same as the number of bytes nor the number of sample points in the sound data chunk (ie, it won't be unless you're dealing with a mono waveform). the total number of sample points in the file is numsampleframes times numchannels.

the samplesize is the number of bits in each sample point. it can be any number from 1 to 32.

the samplerate field is the sample rate at which the sound is to be played back in sample frames per second.

one, and only one, common chunk is required in every form aiff.

---

sound data chunk

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

```
#define sounddataid 'ssnd'  /* chunk id for sound data chunk */

typedef struct {
  id               chunkid;
  long             chunksize;

  unsigned long  offset;
  unsigned long  blocksize;
  unsigned char  waveformdata[];
}  sounddatachunk;
```

the id is always **ssnd**. 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).

you can determine how many bytes of actual waveform data there is by subtracting 8 from the chunksize. remember that the number of sample frames, bit resolution, and other information is gotten from the common chunk.

the offset field determines where the first sample frame in the waveformdata starts. the offset is in bytes. most applications won't use offset and should set it to zero. use for a non-zero offset is explained in "block-aligning waveform data".

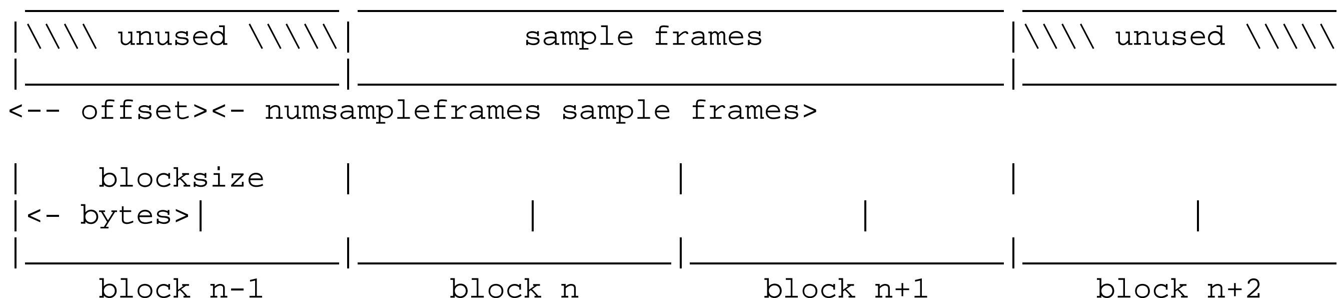
the blocksize is used in conjunction with offset for block-aligning waveform data. it contains the size in bytes of the blocks that waveform data is aligned to. as with offset, most applications won't use blocksize and should set it to zero. more information on blocksize is in "block-aligning waveform data".

the waveformdata array contains the actual waveform data. the data is arranged into what are called *sample frames* the number of sample frames in waveformdata is determined by the numsampleframes field in the common chunk. for more information, see "sample points and sample frames".

the sound data chunk is required unless the numsampleframes field in the common chunk is zero. one, and only one, sound data chunk may appear in a form aiff.

block-aligning waveform data

there may be some applications that, to ensure real time recording and playback of audio, wish to align waveform data into fixed-size blocks. this alignment can be accomplished with the offset and blocksize parameters of the sound data chunk, as shown below.



above, the first sample frame starts at the beginning of block n. this is accomplished by skipping the first offset bytes (ie, some stored pad bytes) of the waveformdata. note that there may also be pad bytes stored at the end of waveformdata to pad it out so that it ends upon a block boundary.

the blocksize specifies the size in bytes of the block to which you would align the waveform data. a blocksize of 0 indicates that the waveform data does not need to be block-aligned. applications that don't care about block alignment should set the blocksize and offset to 0 when creating aiff files. applications that write block-aligned waveform data should set blocksize to the appropriate block size. applications that modify an existing aiff file should try to preserve alignment of the waveform data, although this is not required. if an application does not preserve alignment, it should set the blocksize and offset to 0. if an application needs to realign waveform data to a different sized block, it should update blocksize and offset accordingly.

---

the marker chunk

the marker chunk contains markers that point to positions in the waveform data. markers can be used for whatever purposes an application desires. the instrument chunk, defined later in this document, uses markers to mark loop beginning and end points.

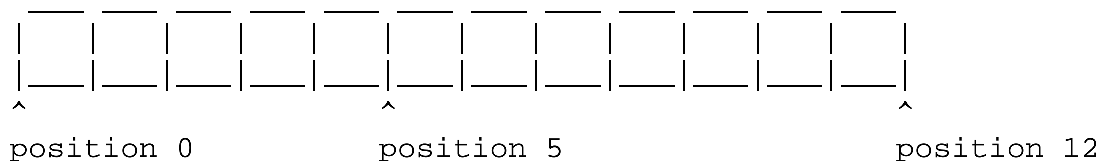
a marker structure is as follows:

```
typedef short  markerid;

typedef struct {
  markerid       id;
  unsigned long  position;
  pstring        markername;
} marker;
```

the id is a number that uniquely identifies that marker within an aiff. the id can be any positive non-zero integer, as long as no other marker within the same form aiff has the same id.

the marker's position in the waveformdata is determined by the position field. markers conceptually fall between two sample frames. a marker that falls before the first sample frame in the waveform data is at position 0, while a marker that falls between the first and second sample frame in the waveform data is at position 1. therefore, the units for position are sample frames, not bytes nor sample points.

sample frames



the markername field is a pascal-style text string containing the name of the mark.

note: some "ea iff 85" files store strings as c-strings (text bytes followed by a null terminating character) instead of pascal-style strings. audio iff uses pstrings because they are more efficiently skipped over when scanning through chunks. using pstrings, a program can skip over a string by adding the string count to the address of the first character. c strings require that each character in the string be examined for the null terminator.

marker chunk format

the format for the data within a marker chunk is shown below.

```
#define markerid 'mark'  /* chunkid for marker chunk */

typedef  struct {
  id               chunkid;
  long             chunksize;

  unsigned short   nummarkers;
  marker           markers[];
} markerchunk;
```

the id is always **mark**. chunksize is the number of bytes in the chunk, not counting the 8 bytes used by id and size fields.

the nummarkers field is the number of marker structures in the marker chunk. if nummarkers is not 0, it is followed by that many marker structures, one after the other. because all fields in a marker structure are an even number of bytes, the length of any marker will always be even. thus, markers are packed together with no unused bytes between them. the markers need not be placed in any particular order.

the marker chunk is optional. no more than one marker chunk can appear in a form aiff.

---

the instrument chunk

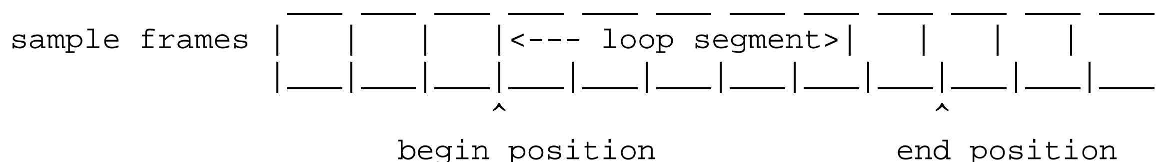
the instrument chunk defines basic parameters that an instrument, such as a midi sampler, could use to play the waveform data.

looping

waveform data can be looped, allowing a portion of the waveform to be repeated in order to lengthen the sound. the structure below describes a loop.

```
typedef struct {
  short     playmode;
  markerid  beginloop;
  markerid  endloop;
} loop;
```

a loop is marked with two points, a begin position and an end position. there are two ways to play a loop, forward looping and forward/backward looping. in the case of forward looping, playback begins at the beginning of the waveform, continues past the begin position and continues to the end position, at which point playback starts again at the begin position. the segment between the begin and end positions, called the loop segment, is played repeatedly until interrupted by some action, such as a musician releasing a key on a musical controller.



with forward/backward looping, the loop segment is first played from the begin position to the end position, and then played backwards from the end position to the begin position. this flip-flop pattern is repeated over and over again until interrupted.

the playmode specifies which type of looping is to be performed:

```
#define nolooping              0
#define forwardlooping         1
#define forwardbackwardlooping 2
```

if nolooping is specified, then the loop points are ignored during playback.

the beginloop is a marker id that marks the begin position of the loop segment.

the endloop marks the end position of a loop. the begin position must be less than the end position. if this is not the case, then the loop segment has 0 or negative length and no looping takes place.

the instrument chunk format

the format of the data within an instrument chunk is described below.

```
#define instrumentid 'inst'  /*chunkid for instruments chunk */

typedef struct {
  id     chunkid;
  long   chunksize;

  char   basenote;
  char   detune;
  char   lownote;
  char   highnote;
  char   lowvelocity;
  char   highvelocity;
  short  gain;
  loop   sustainloop;
  loop   releaseloop;
} instrumentchunk;
```

the id is always **inst**. chunksize should always be 20 since there are no fields of variable length.

the basenote is the 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). units are midi note numbers, and are in the range 0 through 127. middle c is 60.

the detune 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.

the lownote and highnote fields specify the suggested 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 base note does not have to be within this range. units for lownote and highnote are midi note values.

the lowvelocity and highvelocity fields specify the suggested range of 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. units are midi velocity values, 1 (lowest velocity) through 127 (highest velocity).

the gain 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 sustainloop field specifies a loop that is to be played when an instrument is sustaining a sound.

the releaseloop field specifies a loop that is to be played when an instrument is in the release phase of playing back a sound. the release phase usually occurs after a key on an instrument is released.

the instrument chunk is optional. no more than 1 instrument chunk can appear in one form aiff.

---

the midi data chunk

the midi data chunk can be used to store midi data.

the primary purpose of this chunk is to store midi system exclusive messages, although other types of midi data can be stored in the chunk as well. as more instruments come to market, they will likely have parameters that have not been included in the aiff specification. the sys ex messages for these instruments may contain many parameters that are not included in the instrument chunk. for example, a new midi sampler may support more than the two loops per waveform. these loops will likely be represented in the sys ex message for the new sampler. this message can be stored in the midi data chunk (ie, so you have some place to store these extra loop points that may not be used by other instruments).

```
#define mididataid 'midi' /* chunkid for midi data chunk */

typedef struct {
  id              chunkid;
  long            chunksize;

  unsigned char   mididata[];
} mididatachunk;
```

the id is always **midi**. 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.

the mididata field contains a stream of midi data. there should be as many bytes as chunksize specifies, plus perhaps a pad byte if needed.

the midi data chunk is optional. any number of these chunks may exist in one form aiff. if midi system exclusive messages for several instruments are to be stored in a form aiff, it is better to use one midi data chunk per instrument than one big midi data chunk for all of the instruments.

---

the audio recording chunk

the audio recording chunk contains information pertinent to audio recording devices.

```
#define audiorecording id 'aesd'   /* chunkid for audio recording chunk. */

typedef struct {
  id              chunkid
  long            chunksize;

  unsigned char   aeschannelstatusdata[24];
} audiorecordingchunk;
```

the id is always **aesd**. chunksize should always be 24 since there are no fields of variable length.

the 24 bytes of aescchannelstatusdata are specified in the "aes recommended practice for digital audio engineering - serial transmission format for linearly represented digital audio data", transmission of digital audio between audio devices. this information is duplicated in the audio recording chunk for convenience. of general interest would be bits 2, 3, and 4 of byte 0, which describe recording emphasis.

the audio recording chunk is optional. no more than 1 audio recording chunk may appear in one form aiff.

---

the application specific chunk

the application specific chunk can be used for any purposes whatsoever by developers and application authors. for example, an application that edits sounds might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, etc.

```
#define applicationspecificid 'appl' /* chunkid for application specific chunk. */

typedef struct {
  id        chunkid;
  long      chunksize;

  char      applicationsignature[4];
  char      data[];
} applicationspecificchunk;
```

the id is always **appl**. 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.

the applicationsignature field is used by applications which run on platforms from apple computer, inc. for the apple ii, this field should be set to 'pdos'. for the mac, this field should be set to the application's four character signature as registered with apple technical support.

the data field is the data specific to the application. the application determines how many bytes are stored here, and what their purpose are. a trailing pad byte must follow if that is needed in order to make the chunk an even size.

the application specific chunk is optional. any number of these chunks may exist in a one form aiff.

---

the comments chunk

the comments chunk is used to store comments in the form aiff. standard iff has an annotation chunk that can also be used for comments, but this new comments chunk has two fields (per comment) not found in the standard iff chunk. they are a time-stamp for the comment and a link to a marker.

comment structure

a comment structure consists of a time stamp, marker id, and a text count followed by text.

```
typedef struct {
  unsigned long   timestamp;
  markerid        marker;
  unsigned short  count;
  char            text[];
} comment;
```

the timestamp indicates when the comment was created. on the amiga, units are the number of seconds since january 1, 1978. on the mac, units are the number of seconds since january 1, 1904.

a comment can be linked to a marker. this allows applications to store long descriptions of markers as a comment. if the comment is referring to a marker, then the marker field is the id of that marker. otherwise, marker is 0, indicating that this comment is not linked to any marker.

the count is the length of the text that makes up the comment. this is a 16-bit quantity, allowing much longer comments than would be available with a pstring. this count does not include any possible pad byte needed to make the comment an even number of bytes in length.

the text field contains the comment itself, followed by a pad byte if needed to make the text field an even number of bytes.

comments chunk format

```
#define commentid 'comt'  /* chunkid for comments chunk  */

typedef struct {
  id              chunkid;
  long            chunksize;

  unsigned short  numcomments;
  char            comments[];
}commentschunk;
```

the id is always **comt**. chunksize is the number of bytes in the chunk, not counting the 8 bytes used by id and size fields.

the numcomments field contains the number of comment structures in the chunk. this is followed by the comment structures, one after the other. comment structures are always even numbers of bytes in length, so there is no padding needed between structures.

the comments chunk is optional. no more than 1 comments chunk may appear in one form aiff.

---

the text chunks, name, author, copyright, annotation

these four optional chunks are included in the definition of every standard iff file.

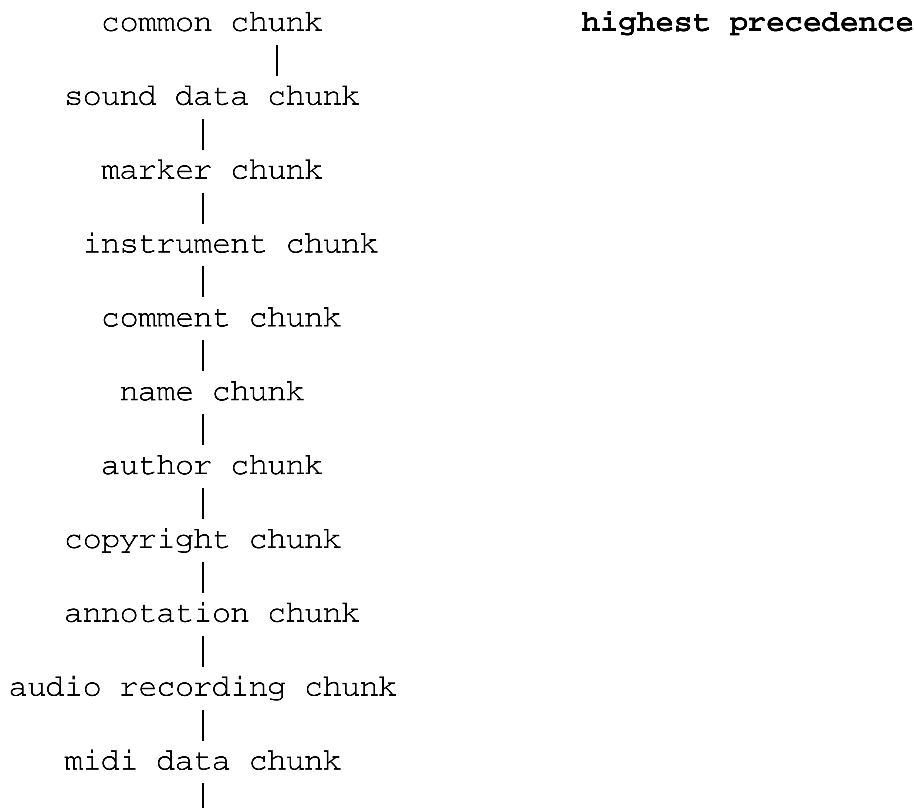
```
#define nameid 'name'   /* chunkid for name chunk */
#define nameid 'auth'   /* chunkid for author chunk */
#define nameid '(c) '   /* chunkid for copyright chunk */
#define nameid 'anno'   /* chunkid for annotation chunk */

typedef struct {
  id      chunkid;
  long    chunksize;
  char    text[];
}textchunk;
```

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.

---

chunk precedence

several of the local chunks for form aiff may contain duplicate information. for example, the instrument chunk defines loop points and some midi sys ex data in the midi data chunk may also define loop points. what happens if these loop points are different? how is an application supposed to loop the sound? such conflicts are resolved by defining a precedence for chunks. this precedence is illustrated below.



```
       common chunk              highest precedence
                |
     sound data chunk
            |
       marker chunk
            |
      instrument chunk
            |
       comment chunk
            |
        name chunk
            |
       author chunk
            |
     copyright chunk
            |
     annotation chunk
            |
  audio recording chunk
            |
     midi data chunk
            |
```

```
application specific chunk       lowest precedence
```

the common chunk has the highest precedence, while the application specific chunk has the lowest. information in the common chunk always takes precedence over conflicting information in any other chunk. the application specific chunk always loses in conflicts with other chunks. by looking at the chunk hierarchy, for example, one sees that the loop points in the instrument chunk take precedence over conflicting loop points found in the midi data chunk.

it is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly (ie, so that conflicts tend not to exist).

---

errata

the apple iigs sampled instrument format also defines a chunk with id of "inst," which is not the same as the aiff instrument chunk. a good way to tell the two chunks apart in generic iff-style readers is by the chunksize fields. the aiff instrument chunk's chunksize field is always 20, whereas the apple iigs sampled instrument format instrument chunk's chunksize field, for structural reasons, can never be 20.

storage of aiff on apple and other platforms

on a macintosh, the form aiff, is stored in the data fork of an audio iff file. the macintosh file type of an audio iff file is 'aiff'. this is the same as the formtype of the form aiff. macintosh applications should not store any information in audio iff file's resource fork, as this information may not be preserved by all applications. applications can use the application specific chunk, defined later in this document, to store extra information specific to their application.

audio iff files may be identified in other apple file systems as well. on a macintosh under mfs or hfs, the form aiff is stored in the data fork of a file with file type "aiff." this is the same as the formtype of the form aiff.

on an operating system such as ms-dos or unix, where it is customary to use a file name extension, it is recommended that audio iff file names use ".aif" for the extension.

referring to audio iff

the official name is "audio interchange file format". if an application needs to present the name of this format to a user, such as in a "save as..." dialog box, the name can be abbreviated to audio iff. referring to audio iff files by a four-letter abbreviation (ie, "aiff") at the user-level should be avoided.

converting extended data to a unsigned long

the sample rate field in a common chunk is expressed as an 80 bit ieee standard 754 floating point number. this isn't a very useful format for computer software and audio hardware that can't directly deal with such floating point values. for this reason, you may wish to use the following convertfloat() hack to convert the floating point value to an unsigned long. this function doesn't handle very large floating point values, but certainly larger than you would ever expect a typical sampling rate to be. this function assumes that the passed *buffer* arg is a pointer to the 10 byte array which already contains the 80-bit floating point value. it returns the value (ie, sample rate in hertz) as an unsigned long. note that the fliplong() function is only of use to intel cpu's which have to deal with aiff's big endian order. if not compiling for an intel cpu, comment out the intel_cpu define.

```
#define intel_cpu
```

```
#ifdef intel_cpu
/* ************************** fliplong() *****************************
 * converts a long in "big endian" format (ie, motorola 68000) to intel
 * reverse-byte format, or vice versa if originally in big endian.
 ******************************************************************** */

void fliplong(unsigned char * ptr)
{
   register unsigned char val;

   /* swap 1st and 4th bytes */
   val = *(ptr);
   *(ptr) = *(ptr+3);
   *(ptr+3) = val;

   /* swap 2nd and 3rd bytes */
   ptr += 1;
   val = *(ptr);
   *(ptr) = *(ptr+1);
   *(ptr+1) = val;
}
#endif

/* ************************* fetchlong() ******************************
 * fools the compiler into fetching a long from a char array.
 ******************************************************************** */

unsigned long fetchlong(unsigned long * ptr)
{
   return(*ptr);
}

/* ************************* convertfloat() *****************************
 * converts an 80 bit ieee standard 754 floating point number to an unsigned
 * long.
 ********************************************************************* */

unsigned long convertfloat(unsigned char * buffer)
{
   unsigned long mantissa;
   unsigned long last = 0;
   unsigned char exp;

#ifdef intel_cpu
   fliplong((unsigned long *)(buffer+2));
#endif

   mantissa = fetchlong((unsigned long *)(buffer+2));
   exp = 30 - *(buffer+1);
   while (exp--)
   {
     last = mantissa;
     mantissa >>= 1;
```

```
   }
   if (last & 0x00000001) mantissa++;
   return(mantissa);
}
```

of course, you may need a complementary routine to take a sample rate as an unsigned long, and put it into a buffer formatted as that 80-bit floating point value.

```
/* ************************ storelong() *****************************
 * fools the compiler into storing a long into a char array.
 ******************************************************************** */

void storelong(unsigned long val, unsigned long * ptr)
{
   *ptr = val;
}

/* ************************* storefloat() ****************************
 * converts an unsigned long to 80 bit ieee standard 754 floating point
 * number.
 ********************************************************************** */

void storefloat(unsigned char * buffer, unsigned long value)
{
   unsigned long exp;
   unsigned char i;

   memset(buffer, 0, 10);

   exp = value;
   exp >>= 1;
   for (i=0; i<32; i++) { exp>>= 1;
      if (!exp) break;
   }
   *(buffer+1) = i;

   for (i=32; i; i--)
   {
      if (value & 0x80000000) break;
      value <<= 1; } storelong(value, buffer+2); #ifdef intel_cpu fliplong((unsigned
long *)(buffer+2)); #endif }
```

multi-sampling

many midi samplers allow splitting up the midi note range into smaller ranges (for example by octaves) and assigning a
different waveform to play over each range. if you wanted to store all of those waveforms into a single data file, what
you would do is create an iff list (or perhaps cat if you wanted to store forms other than aiff in it) and then include an
embedded form aiff for each one of the waveforms. (ie, each waveform would be in a separate form aiff, and all of these
form aiffs would be in a single list file).

content-type: text/html; charset=iso-8859-1; name="new wave riff chunks.htm"
content-disposition: inline; filename="new wave riff chunks.htm"
content-base: "file:///d|/source/sound/format/new%20w
ave%20riff%20chunks.htm"

x-mime-autoconverted: from 8bit to quoted-printable by skynet.usb.ve id taa05017

new wave riff chunks

---

added: 05/01/92
author: microsoft, ibm

most of the information in this section comes directly from the ibm/microsoft riff standard document.

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 , and both of these chunks are mandatory in a wave file.

```
?/ϕοντ>
riff( 'wave'
<'fmt'-ck> // format
[] // fact chunk
[] // cue points
[] // playlist
[] // associated data list
) // wave data
```

the wave chunks are described in the following sections.

fact chunk

the stores file dependent information about the contents of the wave file. this chunk is defined as follows:

```
?/ϕοντ> fact( )
```

<dwsamplelength> represents the length of the data in samples. the <nsamplespersec> field from the wave format header is used in conjunction with the <dwsamplelength> field to determine the length of the data in seconds.

the fact chunk is required for all new wave formats. the chunk is not required for the standard wave_format_pcm files.

the fact chunk will be expanded to include any other information required by future wave formats. added fields will appear following the field. applications can use the chunk size field to determine which fields are present.

cue points chunk

the <cue-ck> cue-points chunk identifies a series of positions in the waveform data stream. the is defined as follows:

```
?/φovτ> cue( // count of cue points
... ) // cue-point table

?/φovτ> struct {
dword dwname;
dword dwposition;
fourcc fccchunk;
dword dwchunkstart;
dword dwblockstart;
dword dwsampleoffset;
}
```

the <cue-point> fields are as follows:

field	description
dwname	specifies the cue point name. each record must have a unique dwname field.
dwposition	specifies the sample position of the cue point. this is the sequential sample number within the play order. see "playlist chunk," later in this document, for a discussion of the play order.
fccchunk	specifies the name or chunk id of the chunk containing the cue point.
dwchunkstart	specifies the position of the start of the data chunk containing the cue point. this should be zero if there is only one chunk containing data (as is currently always the case).
dwblockstart	specifies the position of the start of the block containing the position. this is the byte offset from the start of the data section of the chunk, not the chunk's fourcc.
dwsampleoffset	specifies the sample offset of the cue point relative to the start of the block.

examples of file position values

the following table describes the field values for a wave file containing a single 'data' chunk:

cue point location	field	value
within pcm data	**fccchunk**	fourcc value 'data'.
	dwchunkstart	zero value.
	dwblockstart	file position of the sample (nblockalign aligned bytes) relative to the start of the data section of the 'data' chunk (not the fourcc).
	dwsampleoffset	sample position of the cue point relative to the start of the 'data' chunk.

in all other 'data' chunks	**fccchunk**	fourcc value 'data'.
	dwchunkstart	zero value.
	dwblockstart	file position of the enclosing block relative to the start of the data section of the 'data' chunk (not the fourcc). the software can begin the decompression at this point.
	dwsampleoffset	sample position of the cue point relative to the start of the block.

playlist chunk

the playlist chunk specifies a play order for a series of cue points. the is defined as follows:

?plst(
// count of play segments
...) // play-segment table

?struct {
dword dwname;
dword dwlength;
dword dwloops;
}

the <play-segment> fields are as follows:

field	description
dwname	specifies the cue point name. this value must match one of the names listed in the cue-point table.
dwlength	specifies the length of the section in samples.
dwloops	specifies the number of times to play the section.

associated data chunk

the associated data list provides the ability to attach information like labels to sections of the waveform data stream. the is defined as follows:

```
?/φοντ> list( 'adtl'

// label

// note

} // text with data length
```

?/φoντ> **labl(**

)

?/φoντ> **note(**

)

?/φoντ> **ltxt(**

...)

label and note information

the 'labl' and 'note' chunks have similar fields. the 'labl' chunk contains a label, or title, to associate with a cue point. the 'note' chunk contains comment text for a cue point. the fields are as follows:

field	description
dwname	specifies the cue point name. this value must match one of the names listed in the cue-point table.
data	specifies a null-terminated string containing a text label (for the 'labl' chunk) or comment text (for the 'note' chunk).

text with data length information

the "ltxt" chunk contains text that is associated with a data segment of specific length. the chunk fields are as follows:

field	description
dwname	specifies the cue point name. this value must match one of the names listed in the cue-point table.
dwsamplelength	specifies the number of samples in the segment of waveform data.
dwpurpose	specifies the type or purpose of the text. for example, can specify a fourcc code like 'scrp' for script text or 'capt' for close-caption text.
wcountry	specifies the country code for the text. see "country codes" for a current list of country codes.
wlanguage, wdialect	specify the language and dialect codes for the text. see "language and dialect codes" for a current list of language and dialect codes.
wcodepage	specifies the code page for the text.

inst (instrument) chunk

added: 12/29/92

author: ibm
defined for: wave form

the wave form is nearly the perfect file format for storing a sampled sound synthesizer's samples. bits per sample, sample rate, number of channels, and complex looping can be specified with current wave subchunks, but a sample's pitch and its desired volume relative to other samples cannot. the optional instrument subchunk defined below fills in these needed parameters:

|| ?/φovτ> **inst(**

)

bunshiftednote	the midi note number that corresponds to the unshifted pitch of the sample. valid values range from 0 to 127.
chfinetune	the pitch shift adjustment in cents (or 100ths of a semitone) needed to hit bunshiftednote value exactly. chfinetune can be used to compensate for tuning errors in the sampling process. valid values range from -50 to 50.
chgain	the suggested volume setting for the sample in decibels. a value of zero decibels suggests no change in the volume. a value of -6 decibels suggests reducing the amplitude of the sample by two.
blownote and bhigh note	the suggested usable midi note number range of the sample. valid values range from 0 to 127.
blowvelocity and bhighvelocity	the suggested usable midi velocity range of the sample. valid values range from 0 to 127.

smpl (sample) chunk

added: 11/09/93
author: digidesign, sonic foundary, turtle beach
defined for: wave form

the sampled instrument chunk describes the minimum necessary information needed to allow a sampling keyboard to use a wave file as an instrument. samplers which require more information can save their extended information in the sampler specific data section. the is defined as follows:

|| ?/φovτ> **smpl(**

)

```
struct

{

dword dwidentifier;

dword dwtype;

dword dwstart;

dword dwend;
```

```
dword dwfraction;

dword dwplaycount;

}
```

the chunk:

dwmanufacturer	specifies the mma manufacturer code for the intended target device. the high byte 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 sample is not intended for a specific manufacturer, then this field should be set to zero.
dwproduct	specifies the product code of the intended target device for the dwmanufacturer. if the sample is not intended for a specific manufacturer's product, then this field should be set to zero.
dwsampleperiod	specifies the period of one sample in nanoseconds (normally 1/nsamplespersec from the waveformat structure for the riff wave file--however, this field allows fine tuning). for example, 44.1 khz would be specified as 22675 (0x00005893).
dwmidiunitynote	specifies the midi note which will replay the sample at original pitch. this value ranges from 0 to 127 (a value of 60 represents middle c as defined by the mma).
dwmidipitchfraction	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.
dwsmpteformat	specifies the smpte time format used in the dwsmpteoffset field. possible values are (unrecognized formats should be ignored): 0 - specifies no smpte offset (dwsmpteoffset should also be zero) 24 - specifies 24 frames per second 25 - specifies 25 frames per second 29 - specifies 30 frames per second with frame dropping ('30 drop') 30 - specifies 30 frames per second
dwsmpteoffset	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)].
csampleloops	specifies the number (count) of records that are contained in the chunk. the records are stored immediately following the cbsamplerdata field.
cbsamplerdata	specifies the size in bytes of the optional . sampler specific data is stored imediately following the records. the cbsamplerdata field will be zero if no extended sampler specific information is stored in the chunk.

the structure:

dwidentifier	identifies the unique 'name' of the loop. this field may correspond with a name stored in the chunk. the name data is stored in the chunk.
dwtype	specifies the loop type: 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)
dwstart	specifies the startpoint of the loop in samples.
dwend	specifies the endpoint of the loop in samples (this sample will also be played).
dwfraction	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.
dwplaycount	specifies the number of times to play the loop. a value of 0 specifies an infinite sustain loop.

??