```
  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

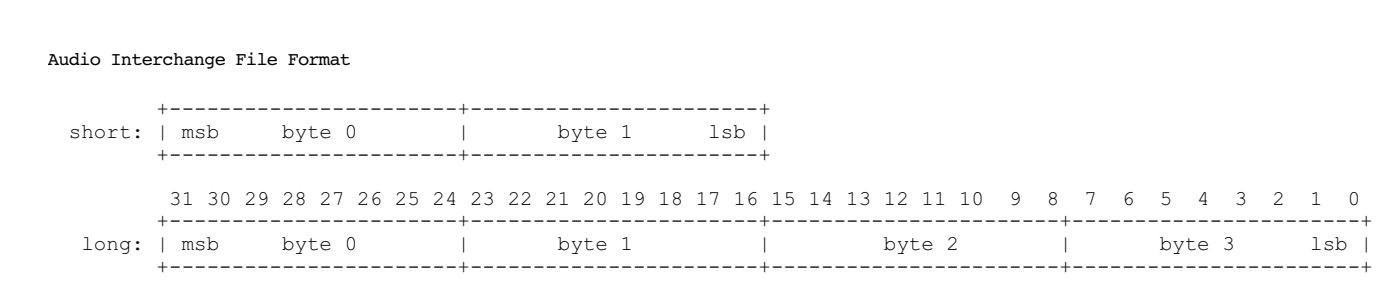
Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

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
```



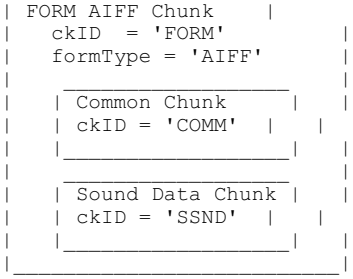
File Structure

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.



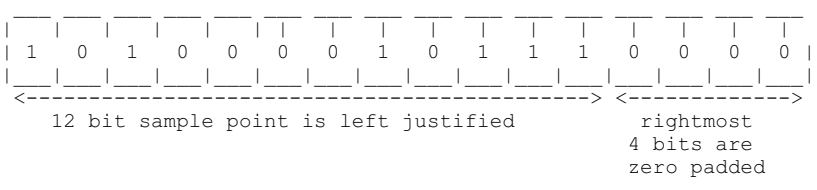
Sample Points and Sample Frames

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.

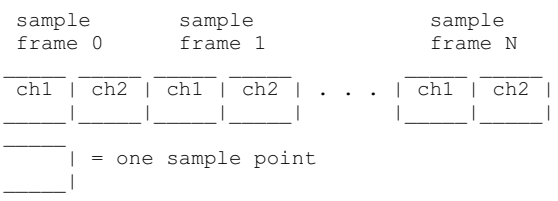
Furthermore, the data bits should be left-justified, with any remaining (ie, pad) bits zeroed. For example, consider the case of a 12-bit sample point. It has 12 bits, so the sample point must be saved as a 16-bit word. Those 12 bits should be left-justified so that they become bits 4 to 15 inclusive, 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).



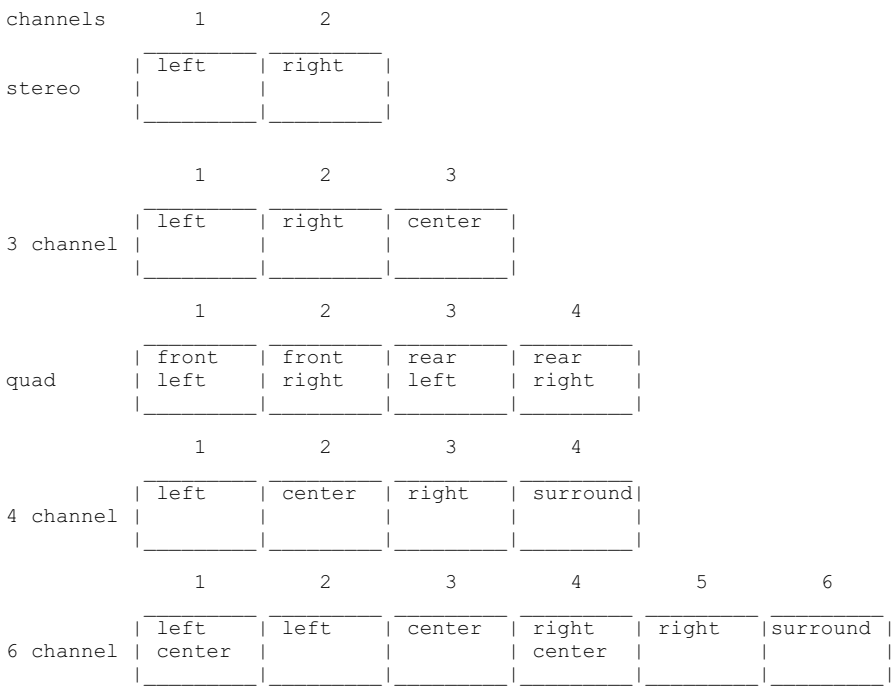
For multichannel sounds (for example, a stereo waveform), single sample points from each channel are interleaved. For example, assume a stereo (ie, 2 channel) waveform. Instead of storing all of the sample points for the left channel first, and then storing all of the sample points for the right channel 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).



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".

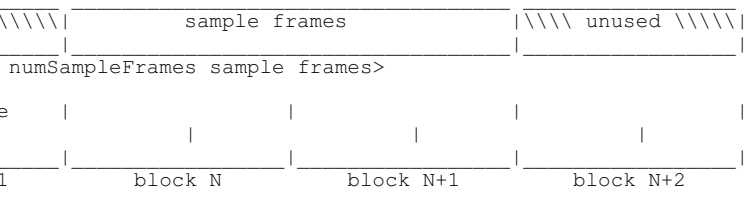
The blockSize is used in conjunction with offset for block-aligning waveform data. It contains the size in bytes of the blocks that waveform data is aligned to. As with offset, most applications won't use blockSize and should set it to zero. More information on blockSize is in "Block-Aligning Waveform 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.

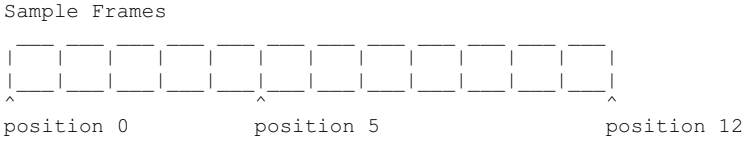
A marker structure is as follows:

```
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.



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

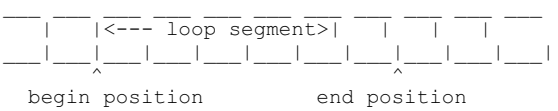
The Instrument Chunk defines basic parameters that an instrument, such as a MIDI sampler, could use to play the waveform data.

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 a 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). See About Interchange File Format for details about LISTs.

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