Creating AIFF Audio Formatted files

Also: Sun .au sound file format, and WAVE sound file format Written by Paul Bourke
September 1996

This document briefly describes the minimal requirements for creating an AIFF formatted sound file of a single channel, it includes some example C source, and finally it contains the complete description of the format for version 1.3 of the specification.

There are only three required sections, a **header**, **common** chunk, and **sound data** chunk.

The **header** simply consists of the letters "FORM", the total size in bytes of the remainder of the file, followed by the letters "AIFF".

The **common** chunk starts with the letters "COMM" followed by the size of the rest of common chunk, 18. The data of the common chunk consists of the number of channels, the number of samples in each channel, the size in bits of the samples, and finally the sampling rate.

The **sound data** chunk starts with the letters "SSND" followed by the size of the rest of the data chunk in bytes. The first two fields of the sound data chunk are the offset and block size which are usually 0. The rest of the sound data chunk consists of the actual sound samples. For multiple channels the samples are in interleaved order.

C Source code

```
/*
   Write an AIFF sound file
   Only do one channel, only support 16 bit.
   Supports sample frequencies of 11, 22, 44KHz (default).
   Little/big endian independent!
*/
void Write_AIFF(FILE *fptr,double *samples,long nsamples,int nfreq)
{
   unsigned short v;
   int i;
   unsigned long totalsize;
   double themin,themax,scale,themid;

   /* Write the form chunk */
   fprintf(fptr,"FORM");
   totalsize = 4 + 8 + 18 + 8 + 2 * nsamples + 8;
   fputc((totalsize & 0xff000000) >> 24,fptr);
   fputc((totalsize & 0x00ff0000) >> 16,fptr);
   fputc((totalsize & 0x0000ff00) >> 8,fptr);
   fputc((totalsize & 0x000000ff),fptr);
```

```
fprintf(fptr,"AIFF");

/* Write the common chunk */
fprintf(fptr,"COMM");
fputc(0,fptr);                          /* Size */
fputc(0,fptr);
fputc(0,fptr);
fputc(18,fptr);
fputc(0,fptr);                          /* Channels = 1 */
fputc(1,fptr);
fputc((nsamples & 0xff000000) >> 24,fptr);   /* Samples */
fputc((nsamples & 0x00ff0000) >> 16,fptr);
fputc((nsamples & 0x0000ff00) >> 8,fptr);
fputc((nsamples & 0x000000ff),fptr);
fputc(0,fptr);                          /* Size = 16 */
fputc(16,fptr);
fputc(0x40,fptr);                         /* 10 byte sampee rate */
if (nfreq == 11025)
   fputc(0x0c,fptr);
else if (nfreq == 22050)
   fputc(0x0d,fptr);
else
   fputc(0x0e,fptr);
fputc(0xac,fptr);
fputc(0x44,fptr);
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);

/* Write the sound data chunk */
fprintf(fptr,"SSND");
fputc((2*nsamples+8 & 0xff000000) >> 24,fptr);/* Size      */
fputc((2*nsamples+8 & 0x00ff0000) >> 16,fptr);
fputc((2*nsamples+8 & 0x0000ff00) >> 8,fptr);
fputc((2*nsamples+8 & 0x000000ff),fptr);
fputc(0,fptr);                           /* Offset    */
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);                           /* Block     */
fputc(0,fptr);
fputc(0,fptr);
fputc(0,fptr);
```

```
  /* Find the range */
  themin = samples[0];
  themax = themin;
  for (i=1;i<nsamples;i++) {
    if (samples[i] > themax)
      themax = samples[i];
    if (samples[i] < themin)
      themin = samples[i];
  }
  if (themin >= themax) {
    themin -= 1;
    themax += 1;
  }
  themid = (themin + themax) / 2;
  themin -= themid;
  themax -= themid;
  if (ABS(themin) > ABS(themax))
    themax = ABS(themin);
  scale = 32760 / (themax);

  /* Write the data */
  for (i=0;i<nsamples;i++) {
    v = (unsigned short)(scale * (samples[i] - themid));
    fputc((v & 0xff00) >> 8,fptr);
    fputc((v & 0x00ff),fptr);
  }
}
```

Audio Interchange File Format: "AIFF"

A Standard for Sampled Sound Files, Version 1.3

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing for the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths.

Audio IFF conforms to the *"EA IFF 85" Standard for Interchange Format Files* developed by Electronic Arts.

Audio IFF is primarily an **interchange** format, although application designers should find it flexible enough to use as a data storage format as well. If an application does choose to use a

different storage format, it should be able to convert to and from the format defined in this document. This will facilitate the sharing of sound data between applications.

Audio IFF is the result of several meetings held with music developers over a period of ten months in 1987-88.

Another "EA IFF 85" sound storage format is *"8SVX" IFF 8-bit Sampled Voice*, by Electronic Arts. "8SVX", which handles 8-bit monaural samples, is intended mainly for storing sound for playback on personal computers. Audio IFF is intended for use with a larger variety of computers, sampled sound instruments, sound software applications, and high fidelity recording devices.

Data types

A C-like language will be used to describe data structures in this document. The data types used are listed below:

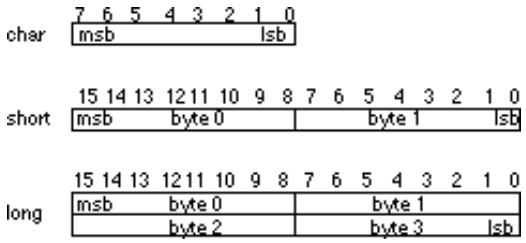
char:	8 bits, signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).
unsigned char:	8 bits, unsigned. Contains any number from zero to 255 (inclusive).
short:	16 bits, signed. Contains any number from -32,768 to 32,767 (inclusive).
unsigned short:	16 bits, unsigned. Contains any number from zero to 65,535 (inclusive).
long:	32 bits, signed. Contains any number from -2,147,483,648 to 2,147,483,647 (inclusive).
unsigned long:	32 bits, unsigned. Contains any number from zero to 4,294,967,295 (inclusive).
extended:	80 bit IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type Extended).
pstring:	Pascal-style string, a one byte count followed by text bytes. The total number of bytes in this data type should be even. A pad byte can be added at the end of the text to accomplish this. This pad byte is not reflected in the count.
ID:	32 bits, the concatenation of four printable ASCII character in the range ' ' (SP, 0x20) through '~' (0x7E). Spaces (0x20) cannot precede printing characters; trailing spaces are allowed. Control characters are forbidden.
OSType:	32 bits. A concatenation of four characters, as defined in Inside Macintosh, vol II.

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. 0x0A12, 0x1, 0x64.

Data Organization

All data is stored in Motorola 68000 format. Data is organized as follows:



Referring to Audio IFF

The official name for this standard is *Audio Interchange File Format*. If an application program 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*.

File Structure

The *"EA IFF 85" Standard for Interchange Format Files* defines an overall structure for storing data in files. Audio IFF conforms to the "EA IFF 85" standard. This document will describe those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85", please refer to the document *"EA IFF 85" Standard for Interchange Format Files*.

An "EA IFF 85" file is made up of a number of *chunks* of data. Chunks are the building blocks of "EA IFF 85" files. A chunk consists of some header information followed by data:

A chunk can be represented using our C-like language in the following manner:

```
typedef struct {
    ID              ckID;       /* chunk ID */
    long            ckSize;     /* chunk Size   */
    char            ckData[];   /* data */
} Chunk;
```

ckID describes the format of the *data* portion a chunk. A program can determine how to interpret the chunk data by examining *ckID*.

ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used

by *ckID* and *ckSize*.

ckData contains the data stored in the chunk. The format of this data is determined by *ckID*. If the data is an odd number of bytes in length, a zero pad byte must be added at the end. The pad byte is not included in *ckSize* .

Note that an array with no size specification (e.g. char ckData[];) indicates a variable-sized array in our C-like language. This differs from standard C.

An Audio IFF file is a collection of a number of different types of chunks. There is a *Common Chunk* which contains important parameters describing the sampled sound, such as it's length and sample rate. There is a *Sound Data Chunk* that contains the actual audio samples. There are several other optional chunks that define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in later sections of this document.

The chunks in a Audio IFF file are grouped together in a container chunk. "EA IFF 85" defines a number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

```
typedef struct {
    ID          ckID;
    long        ckSize;
    ID          formType;
    char        chunks [];
} Chunk;
```

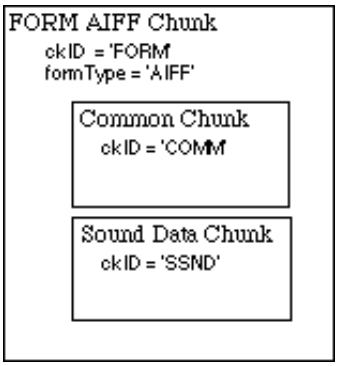
ckID is always 'FORM'. This indicates that this is a FORM chunk.

ckSize contains the size of data portion of the 'FORM' chunk. Note that the data portion has been broken into two parts, *formType* and *chunks*[].

formType describes what's in the 'FORM' chunk. For Audio IFF files, *formType* is always 'AIFF'. This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of *formType* 'AIFF' is called a *FORM AIFF*.

chunks are the chunks contained within the FORM. These chunks are called *local chunks*. A FORM AIFF along with its local chunks make up an Audio IFF file.

Here is an example of a simple Audio IFF file. It consists of a file containing single FORM AIFF which contains two local chunks, a Common Chunk and a Sound Data Chunk.



There are no restrictions on the ordering of local chunks within a FORM AIFF.

On an Apple II, the FORM AIFF is stored in a ProDOS file. The file type is 0xD8 and the aux type is 0x0000. AIFF versions 1.2 and earlier used file type 0xCB, which is incorrect. Please see the Apple II File Type Note for file type 0xD8 and aux type 0x0000 for strategies on dealing with this inconsistency.

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 or Apple II 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.

On an operating system that uses file extensions, such as MS-DOS or UNIX, it is recommended that Audio IFF file names have a ".AIF" extension.

A more detailed example of an Audio IFF file can be found in the Appendix. Please refer to this example as often as necessary while reading the remainder of this document.

Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections. The *ckIDs* for each chunk are also defined.

There are two types of chunks, those that are required and those that are optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has greater than zero length. All other chunks are optional. All applications that use FORM AIFF must be able to read the 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.

Common Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

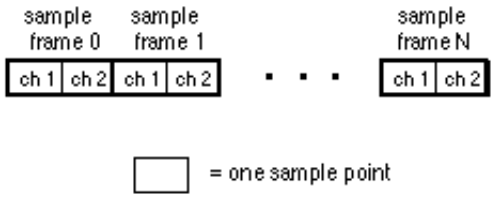
```
#define CommonID    'COMM'  /* ckID for Common Chunk */
```

```
typedef struct {
    ID              ckID;
    long            ckSize;
    short           numChannels;
    unsigned long   numSampleFrames;
    short           sampleSize;
    extended        sampleRate;
} CommonChunk;
```

ckID is always 'COMM'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*. For the Common Chunk, *ckSize* is always 18.

numChannels contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, and 4 means four channel sound, etc. Any number of audio channels may be represented.

The actual sound samples are stored in another chunk, the *Sound Data Chunk*, which will be described shortly. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a *sample frame*. This is illustrated below for the stereo case.



For monophonic sound, a sample frame is a single sample point.

For multichannel sounds, the following conventions should be observed:

	channel 1	2	3	4	5	6
stereo	left	right				
3 channel	left	right	center			
quad	front left	front right	rear left	rear right		
4 channel	left	center	right	surround		
6 channel	left	left center	center	right	right center	surround

numSampleFrames contains the number of sample frames in the *Sound Data Chunk*. Note that *numSampleFrames* is the number of sample frames, not the number of bytes nor the number of sample points in the *Sound Data Chunk*. The total number of sample points in the file is *numSampleFrames* times *numChannels*.

sampleSize is the number of bits in each sample point. It can be any number from 1 to 32. The

format of a sample point will be described in the next section, the *Sound Data Chunk*.

sampleRate 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.

```
#define SoundDataID 'SSND'  /* ckID for Sound Data Chunk */
typedef struct {
    ID                  ckID;
    long                ckSize;
    unsigned long       offset;
    unsigned long       blockSize;
    unsigned char       soundData[];
} SoundDataChunk;
```

ckID is always 'SSND'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

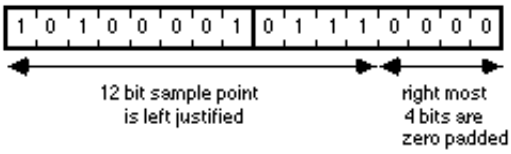
offset determines where the first sample frame in the *soundData* starts. *offset* is in bytes. Most applications won't use *offset* and should set it to zero. Use for a non-zero *offset* is explained in the *Block-Aligning Sound Data* section below.

blockSize is used in conjunction with *offset* for block-aligning sound data. It contains the size in bytes of the blocks that sound 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 the *Block-Aligning Sound Data* section below.

soundData contains the sample frames that make up the sound. The number of sample frames in the *soundData* is determined by the *numSampleFrames* parameter in the *Common Chunk*.

Sample Points

Each sample point in a sample frame is a linear, 2's complement value. The sample points are from 1 to 32 bits wide, as determined by the *sampleSize* parameter in the *Common Chunk*. Sample points are stored in an integral number of contiguous bytes. One to 8 bit wide sample points are stored in one byte, 9 to 16 bit wide sample points are stored in two bytes, 17 to 24 bit wide sample points are stored in 3 bytes, and 25 to 32 bit wide samples are stored in 4 bytes. When the width of a sample point is less than a multiple of 8 bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated below. A 12 bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

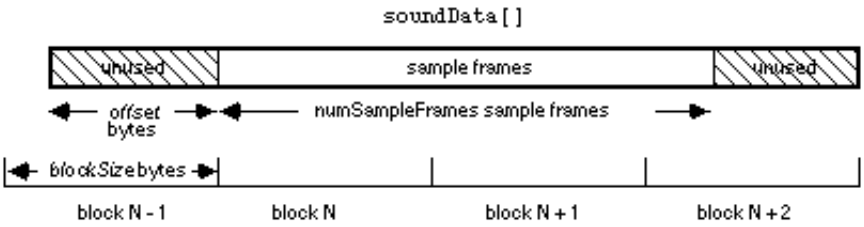


Sample Frames

Sample frames are stored contiguously in order of increasing time. 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.

Block-Aligning Sound Data

There may be some applications that, to insure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This can be accomplished with the *offset* and *blockSize* parameters, as shown below.



Block-aligned sound data

In the above figure, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first *offset* bytes of the *soundData*. Note too that the soundData array can extend beyond valid sample frames, allowing the *soundData* array to end on a block boundary.

blockSize specifies the size in bytes of the block that is to be aligned to. A *blockSize* of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set *blockSize* and *offset* to zero when writing Audio IFF files. Applications that write block-aligned sound data should set *blockSize* to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application doesn't preserve alignment, it should set *blockSize* and *offset* to zero. If an application needs to realign sound data to a different sized block, it should update *blockSize* and *offset* accordingly.

The Sound Data Chunk is required unless the *numSampleFrames* field in the *Common Chunk* is zero. A maximum of one Sound Data Chunk can appear in a FORM AIFF.

Marker Chunk

The Marker Chunk contains markers that point to positions in the sound 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, for example.

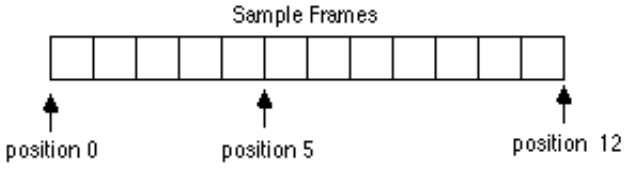
Markers

A marker has the following format.

```
typedef short   MarkerId;
typedef struct {
    MarkerId            id;
    unsigned long       position;
    pstring             markerName;
} Marker;
```

id is a number that uniquely identifies the marker within a FORM 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 sound data is determined by *position* . Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at position zero, while a marker that falls between the first and second sample frame in the sound data is at position 1. Note that the units for position are sample frames, not bytes nor sample points.



markerName 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'  /* ckID for Marker Chunk */
typedef struct {
```

```
    ID                  ckID;
    long                ckSize;
    unsigned short      numMarkers;
    Marker              Markers[];
} MarkerChunk;
```

ckID is always 'MARK'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

numMarkers is the number of markers in the Marker Chunk.

numMarkers, if non-zero, it is followed by the markers themselves. Because all fields in a marker are an even number of bytes in length, 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 ordered in any particular manner.

The Marker Chunk is optional. No more than one Marker Chunk can appear in a FORM AIFF.

Instrument Chunk

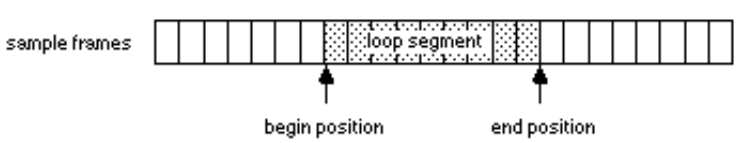
The Instrument Chunk defines basic parameters that an instrument, such as a sampler, could use to play back the sound data.

Looping

Sound data can be looped, allowing a portion of the sound 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 sound, continues past the begin position and continues to the end position, at which point playback restarts again at the begin position. The segment between the begin and end positions, called the *loop segment*, is played over and over again, until interrupted by something, such as the release of a key on a sampling instrument, for example.



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 back to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

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.

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

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 zero or negative length and no looping takes place.

Instrument Chunk Format

The format of the data within an Instrument Chunk is described below.

```
#define InstrumentID    'INST'  /* ckID for Instrument Chunk */
typedef struct {
    ID              ckID;
    long            ckSize;
    char            baseNote;
    char            detune;
    char            lowNote;
    char            highNote;
    char            lowVelocity;
    char            highVelocity;
    short           gain;
    Loop            sustainLoop;
    Loop            releaseLoop;
} InstrumentChunk;
```

ckID is always 'INST'. *ckSize* is the size of the data portion of the chunk, in bytes. For the Instrument Chunk, *ckSize* is always 20.

baseNote is the note at which the instrument plays back the sound data without pitch modification. Units are MIDI (MIDI is an acronym for Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.

detune 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.

lowNote and *highNote* specify the suggested range on a keyboard for playback of the sound data. The sound data should be played if the instrument is requested to play a note between the low and high notes, inclusive. The base note does not have to be within this range. Units for *lowNote* and *highNote* are MIDI note values.

lowVelocity and *highVelocity* specify the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is is between low and high velocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

gain is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0 db means no change, 6 db means double the value of each sample point, while -6 db means halve the value of each sample point.

sustainLoop specifies a loop that is to be played when an instrument is sustaining a sound.

releaseLoop 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 one Instrument Chunk can appear in a FORM AIFF.

MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data (please refer to *Musical Instrument Digital Interface Specification 1.0*, available from the International MIDI Association, for more details on MIDI).

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in this block as well. As more instruments come on the market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the *Instrument Chunk*. For example, a new sampling instrument may have more than the two loops defined in the *Instrument Chunk*. These loops will likely be represented in the MIDI System Exclusive message for the new machine. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

```
#define MIDIDataID  'MIDI'  /* ckID for MIDI Data Chunk */
typedef struct {
    ID                  ckID;
    long                ckSize;
    unsigned char       MIDIdata[];
} MIDIDataChunk;
```

ckID is always ' MIDI'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not

include the 8 bytes used by *ckID* and *ckSize*.

MIDIData contains a stream of MIDI data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a 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.

Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

```
#define AudioRecordingID  'AESD'        /* ckID for Audio Recording */
                                        /*   Chunk.                 */
typedef struct {

    ID                  ckID;
    long                ckSize;
    unsigned char       AESChannelStatusData[24];
} AudioRecordingChunk;
```

ckID is always 'AESD'. *ckSize* is the size of the data portion of the chunk, in bytes. For the Audio Recording Chunk, *ckSize* is always 24.

The 24 bytes of *AESChannelStatusData* are specified in the *AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data*, section 7.1, Channel Status Data. That document describes a format for real-time digital 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 one Audio Recording Chunk may appear in a FORM AIFF.

Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by manufacturers of applications. 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, and the like.

```
#define ApplicationSpecificID  'APPL'   /* ckID for Application */
                                        /*  Specific Chunk.    */
typedef struct {
    ID          ckID;
    long        ckSize;
    OSType      applicationSignature;
    char        data[];
```

```
} ApplicationSpecificChunk;
```

ckID is always 'APPL'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

applicationSignature identifies a particular application. For Macintosh applications, this will be the application's four character signature. For Apple II applications, *applicationSignature* should always be 'pdos', or the hexadecimal bytes 0x70646F73. If *applicationSignature* is 'pdos', the beginning of the data area is defined to be a Pascal-style string (a length byte followed by ASCII string bytes) containing the name of the application. This is necessary because Apple II applications do not have a four-byte signature as do Macintosh applications.

data is the data specific to the application.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EA IFF 85" has an *Annotation Chunk* that can be used for comments, but the Comments Chunk has two features not found in the "EA IFF 85" chunk. They are: 1) a time-stamp for the comment; and 2) a link to a marker.

Comment

A comment 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;
```

timeStamp indicates when the comment was created. Units are the number of seconds since January 1, 1904. (This time convention is the one used by the Macintosh. For procedures that manipulate the time stamp, see The Operating System Utilities chapter in *Inside Macintosh, vol II*). For a routine that will convert this to an Apple II GS/OS format time, please see Apple II File Type Note for filetype 0xD8, aux type 0x0000.

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 *marker* is the ID of that marker. Otherwise, *marker* is zero, indicating that this comment is not linked to a marker.

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.

text contains the comment itself. This text must be padded with a byte at the end to insure that it is an even number of bytes in length. This pad byte, if present, is not included in *count*.

Comments Chunk Format

```
#define CommentID       'COMT'  /* ckID for Comments Chunk.  */
typedef struct {
    ID                  ckID;
    long                ckSize;
    unsigned short      numComments;
    Comment             comments[];
} CommentsChunk;
```

ckID is always ' COMT'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

numComments contains the number of comments in the Comments Chunk. This is followed by the comments themselves. Comments are always an even number of bytes in length, so there is no padding between comments in the Comments Chunk.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

Text Chunks - Name, Author, Copyright, Annotation

These four chunks are included in the definition of every "EA IFF 85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

```
#define NameID          'NAME'  /* ckID for Name Chunk.  */
#define AuthorID        'AUTH'  /* ckID for Author Chunk.  */
#define CopyrightID     '(c) '  /* ckID for Copyright Chunk.  */
#define AnnotationID    'ANNO'  /* ckID for Annotation Chunk.  */
typedef struct {
    ID                  ckID;
    long                ckSize;
    char                text[];
} TextChunk;
```

ckID is either ' NAME', ' AUTH', '(c) ', or ' ANNO', depending on whether the chunk as a Name Chunk, Author Chunk, Copyright Chunk, or Annotation Chunk, respectively. For the Copyright Chunk, the 'c' is lowercase and there is a space (0x20) after the close parenthesis.

ckSize is the size of the data portion of the chunk, in this case the *text*.

text contains pure ASCII characters. It is not a pstring nor a C string. The number of characters in *text* is determined by *ckSize*. The contents of *text* depend on the chunk, as described below:

Name Chunk

text contains the name of the sampled sound. The Name Chunk is optional. No more than one Name Chunk may exist within a FORM AIFF.

Author Chunk

text contains one or more author names. An author in this case is the creator of a sampled sound. The Author Chunk is optional. No more than one Author Chunk may exist within a FORM AIFF.

Copyright Chunk

The Copyright Chunk contains a copyright notice for the sound. text contains a date followed by the copyright owner. The chunk ID '(c) ' serves as the copyright characters '©'. For example, a Copyright Chunk containing the text "1988 Apple Computer, Inc." means "© 1988 Apple Computer, Inc."

The Copyright Chunk is optional. No more than one Copyright Chunk may exist within a FORM AIFF.

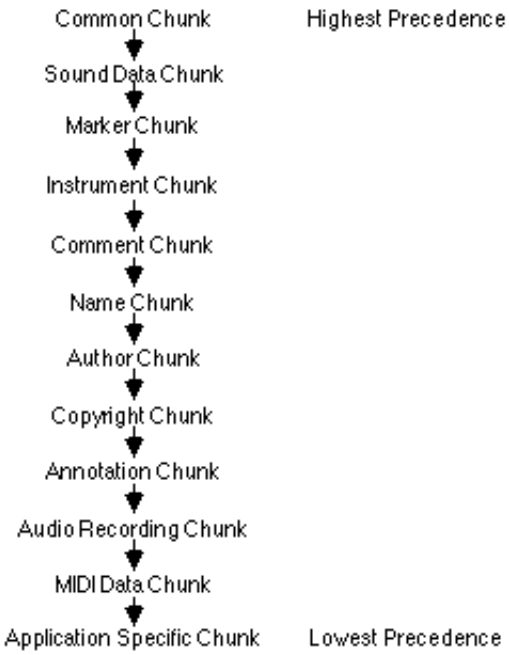
Annotation Chunk

text contains a comment. Use of this chunk is discouraged within FORM AIFF. The more powerful *Comments Chunk* should be used instead. The Annotation Chunk is optional. Many Annotation Chunks may exist within a FORM AIFF.

Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the *Instrument Chunk* defines loop points and MIDI system exclusive 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:

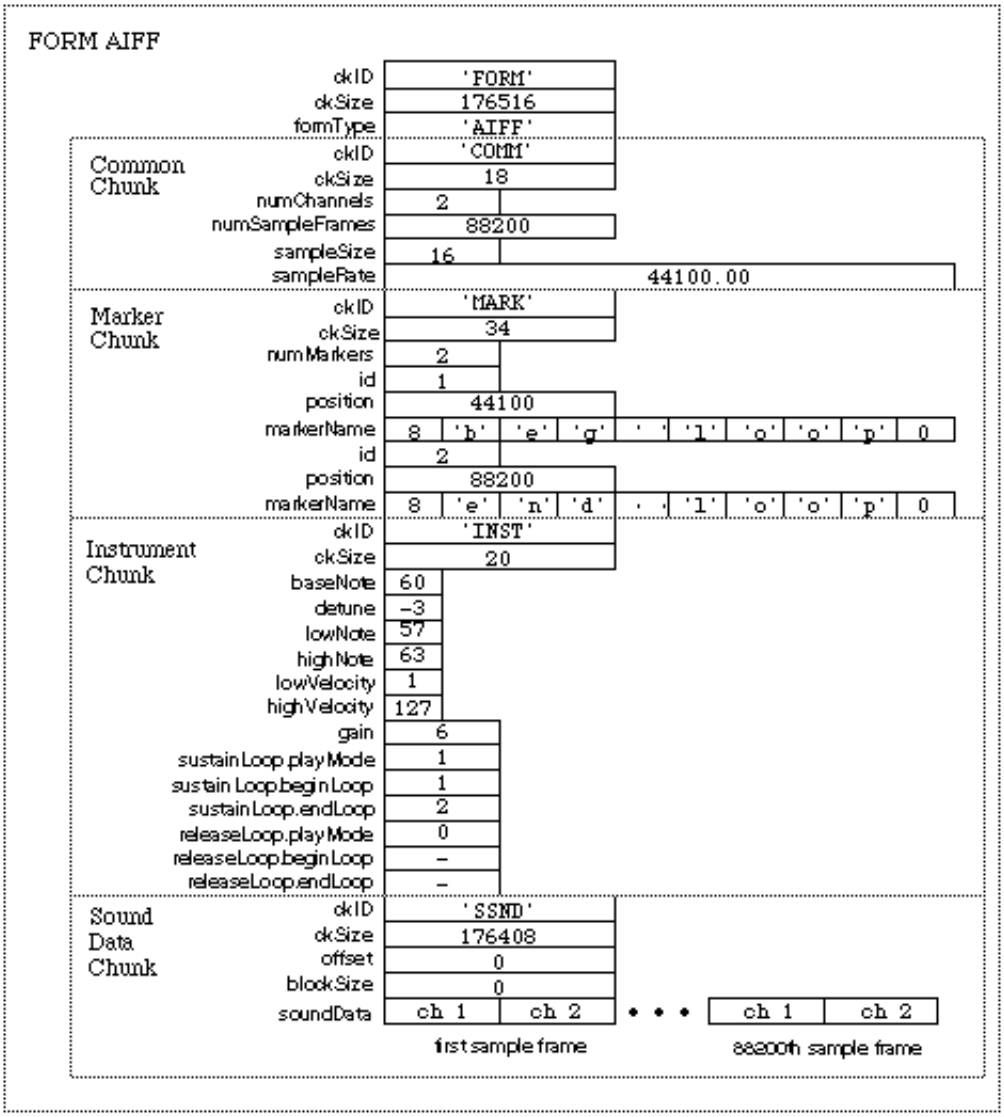


The *Common Chunk* has the highest precedence, while the *Application Specific Chunk* has the lowest. Information in the *Common Chunk* always takes precedence over conflicting information in any other chunk. The *Application Specific Chunk* always loses in conflicts with other chunks. By looking at 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.

Appendix

Illustrated below is an example of a FORM AIFF. An Audio IFF file is simply a file containing a single FORM AIFF. On a Macintosh, the FORM AIFF is stored in the data fork of a file and the file type is 'AIFF'.



Sun .au sound file format

Written by Paul Bourke
November 2001

The .au file format, originally by SUN, is (fortunately) a very straightforward audio format, unfortunately it isn't widely supported outside the UNIX community. The file format is split into three parts....

- A header containing basic information such as the length, number of channels, sample frequency, and data format.
- A variable length informational field. This is designed for copyright information, authors name, etc.
- The audio data which may be stored in a number of formats.

Header

The header is a total of six 4 byte quantities. The description of each 4 byte word is described below.

- A "magic" identification word of ".snd", otherwise known as 0x2e736e64.
- A data offset (bytes) to the audio data. If there is no information section then this would be 24, the size of the remainder of this header.
- The data size (bytes) of the audio data. Since this can be worked out knowing the file length and the data offset above, it is permitted to set this to 0xffffffff.
- The encoding type used for the data, a number from 1, 2, 3, 4, 5, 6, 7, 23, 24, 25, 26, 27. See later.
- The sampling frequency in Hz. The most commonly used frequencies are 11025, 16000, 22050, 32000, 44100, and 48000 Hz.
- The number of channels. For more than one channel the samples are interleaved.

The header field names from SUN are as follows.

```
typedef struct {
   u_32 magic;       /* magic number */
   u_32 hdr_size;    /* size of this header */
   u_32 data_size;   /* length of data (optional) */
   u_32 encoding;    /* data encoding format */
   u_32 sample_rate; /* samples per second */
   u_32 channels;    /* number of interleaved channels */
} Audio_filehdr;
```

Information

This can be any information at all, it automatically starts 24 bytes from the start of the file. If the data offset is 0 then this section is empty. The length of this section is calculated by subtracting 24 from the data offset field in the header.

Audio samples

The audio samples can be encoded in a number of formats, the exact format is described in the 4th word of the header. The options are:

- 1 = 8 bit ISDN u-law
- 2 = 8 bit linear PCM
- 3 = 16 bit linear PCM

- 4 = 24 bit linear PCM
- 5 = 32 bit linear PCM
- 6 = 32 bit IEEE floating point
- 7 = 64 bit IEE floating point
- 23 = 4 bit CCITT G721 ADPCM
- 24 = CCITT G722 ADPCM
- 25 = CCITT G723 ADPCM
- 26 = 5 bit CCITT G723 ADPCM
- 27 = 8 bit ISDN a-law

Simple C source code

The following source stores a time series in the AU format as 16 bit samples, one channel.

```
/*
   Write an AU (Sun) sound file
   Only do one channel, only support 16 bit.
   Supports any (reasonable) sample frequency
   Little/big endian independent!
*/
void Write_AU(FILE *fptr,double *samples,long nsamples,int nfreq)
{
   unsigned short v;
   int i;
   unsigned long totalsize;
   double themin,themax,scale,themid;

   /* Write the form chunk */
   fprintf(fptr,".snd");
   fputc(0,fptr);                                  /* Data offset */
   fputc(0,fptr);
   fputc(0,fptr);
   fputc(24,fptr);
   totalsize = 2 * nsamples;
   fputc((totalsize & 0xff000000) >> 24,fptr);  /* Data size, octets */
   fputc((totalsize & 0x00ff0000) >> 16,fptr);
   fputc((totalsize & 0x0000ff00) >> 8,fptr);
   fputc((totalsize & 0x000000ff),fptr);
   fputc(0,fptr);                                  /* Encoding, 16 PCM */
   fputc(0,fptr);
   fputc(0,fptr);
   fputc(3,fptr);
   fputc((nfreq & 0xff000000) >> 24,fptr);      /* Sample frequency (Hz) */
   fputc((nfreq & 0x00ff0000) >> 16,fptr);
```

```
   fputc((nfreq & 0x0000ff00) >> 8,fptr);
   fputc((nfreq & 0x000000ff),fptr);
   fputc(0,fptr);                                  /* Channels */
   fputc(0,fptr);
   fputc(0,fptr);
   fputc(1,fptr);

   /* Find the range */
   themin = samples[0];
   themax = themin;
   for (i=1;i<nsamples;i++) {
      if (samples[i] > themax)
         themax = samples[i];
      if (samples[i] < themin)
         themin = samples[i];
   }
   if (themin >= themax) {
      themin -= 1;
      themax += 1;
   }
   themid = (themin + themax) / 2;
   themin -= themid;
   themax -= themid;
   if (ABS(themin) > ABS(themax))
      themax = ABS(themin);
   scale = 32760 / (themax);

   /* Write the data */
   for (i=0;i<nsamples;i++) {
      v = (unsigned short)(scale * (samples[i] - themid));
      fputc((v & 0xff00) >> 8,fptr);
      fputc((v & 0x00ff),fptr);
   }
}
```

WAVE sound file format

Written by Paul Bourke
November 2001

WAVE audio files are one of the common formats used to store and play audio data. They support variable sampling frequencies, multiple channels, and a number of compression algorithms. The following gives the minimal requirements necessary to save audio data in this format, it doesn't address compression and only considers sampled audio data.

A WAVE file consists of a number of "chunks", each of these chunks includes an identifier, the size of the chunk in bytes, and any data associated with the chunk. There are two chunks that

are required in order to successfully save sampled audio waveforms, they are a format chunk, and the sample data chunk The main advantage of using this chunk structure is that when parsing a WAVE file you don't need to interpret every chunk type but can skip over the ones you don't need or don't understand.

RIFF header.

The header consists of the characters "RIFF" followed by the size of the rest file, followed by the characters "WAVE" indicating the file type. After this the file contains chunks, while there are a number of potential chunks only the minimum are considered below, a format and data chunk. In general chunks can appear in any order but the format chunk must appear before the data chunk, since it contains information required for the successful interpretation of the data.

Format chunk.

The format chunk has a chunk identifier of the 4 characters "fmt " (note space at end) followed by the following fields.

- chunk size (4 bytes, unsigned long)

  This does not include the 4 bytes of the chunk identifier or the 4 bytes of the chunk size....ie: it is the remaining size of the chunk. For the simple uncompressed WAVE file discussed here this will always be 16 bytes.

- format tag (2 bytes, unsigned long)

  For uncompressed data this field is 1.

- number of channels (2 bytes, unsigned long)

  This contains the number of channels, 1 for mono, 2 for stereo.... Any number of channels are supported although not all systems will make sense of them.

- samples per second (4 bytes, unsigned long)

  This is the sampling frequency in Hz that the waveform is to be played at. This may be any frequency but the standards are 11025, 22050, and 44100 Hz.

- average bytes per second (4 bytes, unsigned long)

  This is a hint for players so they can determine buffering requirements. It is common to attempt to buffer one second of all channels.

- block alignment (2 bytes, unsigned short)

  The block alignment is the storage (bytes) required for one time step, that is, the number

of channels times the sample width in bytes.

- bits per sample (2 bytes, unsigned short)

  This is the bits used to represent each sample, in this example it will be 16 but others are supported. The samples for 16 bit sample is a signed short integer.

Sample Data chunk.

The sample data chunk contain the actual samples of the waveform, it starts with a chunk identifier of "data" followed by the chunk size in bytes, and finally the samples themselves. As before the chunk size does not include the 4 bytes for the identifier or the 4 bytes for the chunk size. Chunks must end on even boundaries and the chunk size also does not include any padding to make this so. The actual number of samples is the chunk size divided by the block alignment setting in the format chunk. Multiple samples are generally interleaved.

Simple C source code

```
/*
   Write an WAVE sound file
   Only do one channel, only support 16 bit.
   Supports any (reasonable) sample frequency
   Little/big endian independent!
*/
void Write_WAVE(FILE *fptr,double *samples,long nsamples,int nfreq)
{
   unsigned short v;
   int i;
   unsigned long totalsize,bytespersec;
   double themin,themax,scale,themid;

   /* Write the form chunk */
   fprintf(fptr,"RIFF");
   totalsize = 2 * nsamples + 36;
   fputc((totalsize & 0x000000ff),fptr);          /* File size */
   fputc((totalsize & 0x0000ff00) >> 8,fptr);
   fputc((totalsize & 0x00ff0000) >> 16,fptr);
   fputc((totalsize & 0xff000000) >> 24,fptr);
   fprintf(fptr,"WAVE");
   fprintf(fptr,"fmt ");                          /* fmt_ chunk */
   fputc(16,fptr);                                /* Chunk size */
   fputc(0,fptr);
   fputc(0,fptr);
   fputc(0,fptr);
   fputc(1,fptr);                                 /* Format tag - uncompressed */
   fputc(0,fptr);
   fputc(1,fptr);                                 /* Channels */
   fputc(0,fptr);
   fputc((nfreq & 0x000000ff),fptr);              /* Sample frequency (Hz) */
   fputc((nfreq & 0x0000ff00) >> 8,fptr);
   fputc((nfreq & 0x00ff0000) >> 16,fptr);
```

```
   fputc((nfreq & 0xff000000) >> 24,fptr);
   bytespersec = 2 * nfreq;
   fputc((bytespersec & 0x000000ff),fptr);      /* Average bytes per second */
   fputc((bytespersec & 0x0000ff00) >> 8,fptr);
   fputc((bytespersec & 0x00ff0000) >> 16,fptr);
   fputc((bytespersec & 0xff000000) >> 24,fptr);
   fputc(2,fptr);                               /* Block alignment */
   fputc(0,fptr);
   fputc(16,fptr);                              /* Bits per sample */
   fputc(0,fptr);
   fprintf(fptr,"data");
   totalsize = 2 * nsamples;
   fputc((totalsize & 0x000000ff),fptr);        /* Data size */
   fputc((totalsize & 0x0000ff00) >> 8,fptr);
   fputc((totalsize & 0x00ff0000) >> 16,fptr);
   fputc((totalsize & 0xff000000) >> 24,fptr);

   /* Find the range */
   themin = samples[0];
   themax = themin;
   for (i=1;i<nsamples;i++) {
      if (samples[i] > themax)
         themax = samples[i];
      if (samples[i] < themin)
         themin = samples[i];
   }
   if (themin >= themax) {
      themin -= 1;
      themax += 1;
   }
   themid = (themin + themax) / 2;
   themin -= themid;
   themax -= themid;
   if (ABS(themin) > ABS(themax))
      themax = ABS(themin);
   scale = 32760 / (themax);

   /* Write the data */
   for (i=0;i<nsamples;i++) {
      v = (unsigned short)(scale * (samples[i] - themid));
      fputc((v & 0x00ff),fptr);
      fputc((v & 0xff00) >> 8,fptr);
   }
}
```