Apple II
File Type Notes

Developer Technical Support

File Type: $D8 (216)
Auxiliary Type: $0000

Full Name: Audio Interchange File Format File
Short Name: Audio IFF File

Revised by: Matt Deatherage March 1991
Written by: Matt Deatherage March 1989

Files of this type and auxiliary type contain sampled sounds in Apple Computer's Audio Interchange File Format (Audio IFF).

Changes since March 1989: Corrected the description of the sample format and added a warning about interchange format testing. Added a reference to `ConvSeconds` for time utilities.

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

This Note describes version 1.3 (January 4, 1989) of the Audio Interchange File Format. This Note describes Audio IFF as it pertains to Apple II developers. For a copy of the original Audio IFF specification, you may order "Audio Interchange File Format v. 1.3" from APDA.

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 ability to convert 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 during 1987 and 1988. Apple Computer greatly appreciates the comments and cooperation provided by all developers who helped define this standard.

Another "EA IFF 85" sound storage format is *"8SVX" IFF 8-bit Sampled Voice*, by Electronic Arts. "8SVX," which handles eight-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.

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. Although the Apple IIGS Sampled Instrument format is often abbreviated as "ASIF," referring to Audio IFF files by a four-letter abbreviation (i.e., "AIFF") in user-level documentation or program-generated messages should be avoided.

The Chunk Concept

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 Note describes those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85," please refer to *"EA IFF 85" Standard for Interchange Format Files*.

Audio IFF, like all IFF-style storage formats, is a series of discrete pieces, or "chunks." Each chunk has an eight-byte "header," which is as follows:

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be ASCII characters in the range $20-$7F. Spaces may not precede printing characters, although trailing spaces are allowed. Characters outside the range $20-$7F are forbidden. A program can determine how to interpret the chunk data by examining ckID.
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
ckData	**Chunk**	The data, specific to each individual chunk. There are exactly ckSize bytes of data here. If the length of the chunk is odd, a pad byte of $00 must be added at the end. The pad byte is not included in ckSize.

Since Audio IFF is primarily an interchange format, it will come as no surprise to find that all constants, such as each chunk's ckSize field, are stored in reverse format (the bytes of multiple-byte values are stored with the high-order bytes first). This is true for all constants, which are marked in their individual descriptions by the **Reverse** notation.

Note: All numeric values in this Note are **signed** unless otherwise noted. This is different from the normal File Type Note convention.

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 its length and sample rate. There is a Sound Data Chunk which contains the actual audio samples. There are several other optional chunks which define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in this Note.

File Structure

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

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`FORM`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first. Also note that the data portion of the chunk is broken into two parts, formType and chunks.
formType	**4 Bytes**	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	**Bytes**	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.

Figure 1 is a pictorial representation of a simple Audio IFF file. It consists of a single `FORM AIFF` which contains two local chunks, a Common Chunk, and a Sound Data Chunk.

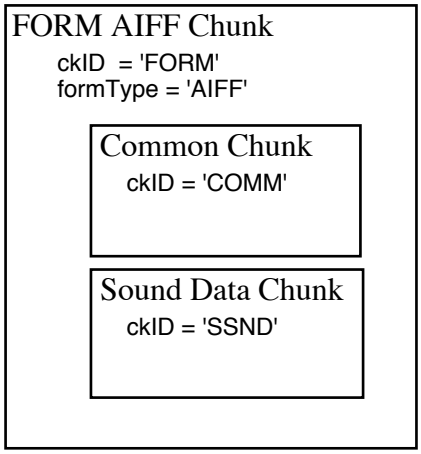

Figure 1–Simple Audio IFF File

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

The `FORM AIFF` is stored in a file with file type $D8 and auxiliary type $0000. Versions 1.2 and earlier of the Audio IFF standard used file type $CB and auxiliary type $0000. This is incorrect; the assignment listed in this Note is the correct assignment. Applications which use Audio IFF files with the older assignment should not perform adversely, since no one should be creating files of any kind with the older assignment. However, we strongly urge developers to update their applications as soon as possible to only create Audio IFF files with file type $D8 and auxiliary type $0000.

Audio IFF files may be identified in other 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`.

Note: Applications should not store any data in the resource fork of an Audio IFF file, since this information may not be preserved by all applications or in translation to foreign file systems. Applications can use the Application Specific Chunk, described later in this Note, to store extra information specific to their application.

In file systems that use file extensions, such as MS-DOS or UNIX, it is recommended that Audio IFF file names have the extension ".AIF."

A more detailed visual example of an Audio IFF file may be found later in this Note. Please refer to it as often as necessary while reading the remainder of this Note.

Local Chunk Types

The formats of the different local chunk types found within a `FORM AIFF` are described in the following sections, as are their ckIDs.

There are two types of chunks: required and optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has a length greater than zero. 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 the chunks in the `FORM AIFF`, even those it chooses not to interpret the optional chunks.

To ensure that this standard remains usable by all developers across machine families, only Apple Computer, Inc. should define new chunk types for `FORM AIFF`. If you have suggestions for new chunk types, Apple is happy to listen. Please send all comments to the address listed in "About File Type Notes" to the attention of Audio IFF Suggestions.

The Common Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`COMM`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. For the Common Chunk, this is always 18.
numChannels	**Rev. Word**	The number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, 4 means four-channel sound, and so on. Any number of audio channels may be represented. The actual sounds samples are stored in the Sound Data Chunk.
numSampleFrames	**Rev. Unsigned Long**	The number of sample frames in the Sound Data Chunk. Sample frames are described below. Note that numSampleFrames is the number of sample frames, not the

		number of bytes nor the number of sample points (also described below) in the Sound Data Chunk. The total number of sample points in the file is numSampleFrames multiplied by numChannels.
sampleSize	**Rev. Word**	The number of bits in each sample point. This can be any number from 1 to 32.
sampleRate	**Rev. Extended**	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`.

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 linear, two's-complement value representing a sample of a sound at a given point in time. A sample point may be from 1 to 32 bits wide, as determined by sampleSize in the Common Chunk. Sample points are stored in an integral number of contiguous bytes. One- to eight-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 three bytes, and 25- to 32-bit wide sample points are stored in four bytes (most significant byte first). When the width of a sample point is not a multiple of eight bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated in Figure 2. A 12-bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

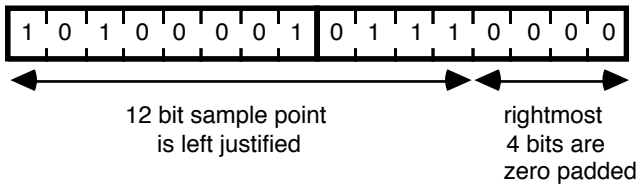

Figure 2–A 12-Bit Sample Point

Warning: Previous versions of this Note omitted the important phrase "two's-complement" in the preceding description of a sample point. This means that sound files created using only this Note (and not also the Audio IFF Specification version 1.3) are incompatible with Audio IFF applications created from the specification. However, any simple amount of testing with Audio IFF files available from a variety of Macintosh programs would instantly make this obvious to an Audio IFF programmer.

We cannot emphasize enough the importance of cross-platform testing with Audio IFF or **any** interchange file format. It's not enough to create the files; you have to make sure your application can read the files created by **other** applications and vice-versa. An interchange format that doesn't interchange correctly is somewhat less than useful.

Sample frames are sets of sample points which are interleaved for multichannel sound. Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each channel available. This is illustrated in Figure 3 for the stereo (two channel) case.

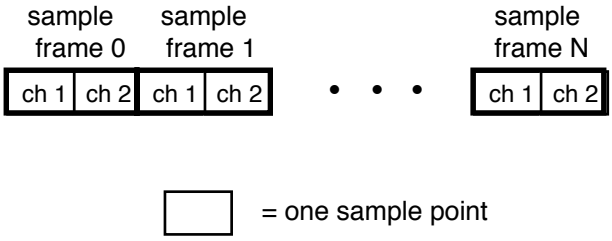

Figure 3–Sample Frames for Multichannel Sound

For monophonic sound, a sample frame is a single sample point. For multichannel sounds, you should follow the conventions in Figure 4.

	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

Figure 4–Sample Frame Conventions for Multichannel Sound

Note: Portions of Figure 4 do not follow the Apple IIGS standard of right on even channels and left on odd channels. The portions that do follow this convention usually use channel two for right instead of channel zero as most Apple IIGS standards. Be prepared to interpret data accordingly.

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.

The Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`SSND`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
offset	**Rev. Unsigned Long**	Determines where the first sample frame in the soundData starts, in bytes. Most applications will not use offset and should set it zero. Use for a non-zero offset is explained below.
blockSize	**Rev. Unsigned Long**	Used in conjunction with offset for block-aligning sound data. It contains the size in bytes of the blocks to which soundData is aligned. As with offset, most applications will not use blockSize and should set it to zero. More information on blockSize is given below.
soundData	**Bytes**	Contains the actual 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.

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

Block-Aligning Sound Data

There may be some applications that, to ensure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This alignment can be accomplished with the offset and blockSize parameters of the Sound Data Chunk, as shown in Figure 5.

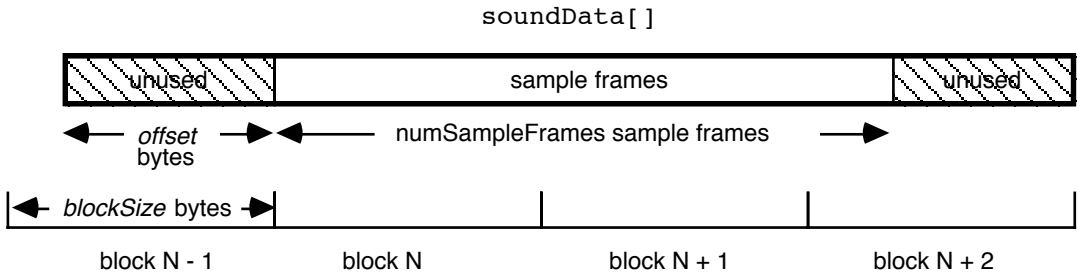

Figure 5–Block-Aligned Sound Data

In Figure 5, 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 bytes can extend beyond valid sample frames, allowing the soundData bytes to end on a block boundary as well.

The blockSize specifies the size in bytes of the block to which you would align the sound data. 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 the blockSize and offset to zero when creating 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 does not preserve alignment, it should set the 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 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 Note, uses markers to mark loop beginning and end points.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`MARK`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
numMarkers	**Rev. Unsigned Word**	The number of markers (defined below) in the Marker Chunk. If non-zero, this is followed by the markers themselves. Because all fields in a marker 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, although the markers themselves need not be ordered in any particular manner.
Marker	**Markers**	Defined below.

A marker has the following format:

MarkerID	**Rev. Word**	The ID for this marker. This is a number that uniquely identifies the marker within a `FORM AIFF`. The number can be any positive, non-zero integer, as long as no other marker within the same `FORM AIFF` has the same ID.

position	**Rev. Unsigned Long**	Determines the marker's position in the sound data. 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 one. Units for position are sample frames, not bytes nor sample points.
markerName	**String**	Pascal-type string containing the name of the mark.

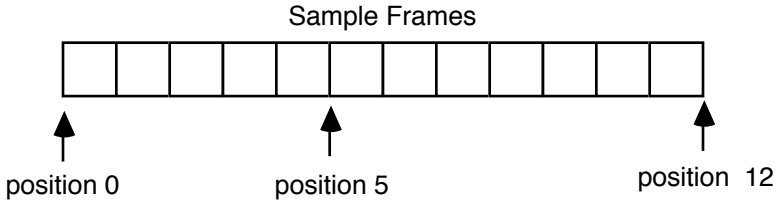

Figure 6–Sample Frame Marker Positions

Note: Some "EA IFF 85" files store strings as C-style strings (null terminated). Audio IFF uses Pascal-style (length byte) strings because they are easier to skip over when scanning a file or a chunk.

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 sample, could use to play the sound data.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`INST`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. For the Instrument Chunk, this field is always 20.
baseNote	**Byte**	The note at which the instrument plays the sound data without pitch modification. Units are MIDI (Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.
detune	**Byte**	Determines how much the instrument should alter the pitch of the sound when it is played . Units are 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	**Byte**	Suggested lowest note 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 lowNote and highNote, inclusive. The base note does not have to be within this range. Units for lowNote and highNote are MIDI note values.
highNote	**Byte**	Suggested highest note on a keyboard for playback of the sound data. See the description of lowNote above.
lowVelocity	**Byte**	The low end of the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is between lowVelocity and highVelocity,

		inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).
highVelocity	**Byte**	The high end of the suggested range of velocities for playback of the sound data. See the description of lowVelocity above.
gain	**Rev. Word**	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	**Loop**	A loop that is to be played when an instrument is sustaining a sound. The format of loops is described below.
releaseLoop	**Loop**	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 format of loops is described below.

Loops

Sound data can be looped, allowing a portion of the sound to be repeated to lengthen the sound. 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 starts again at the begin position. The segment between the begin and end positions, called the loop segment, is played repeatedly until interrupted by a user action, such as the release of a key on a sampling instrument.

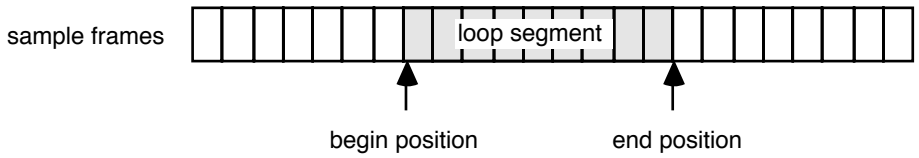

Figure 7–Sample Frame Looping

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.

To end a loop, finish the current loop section and don't repeat it any more. This usually means playing to the end position, but it can mean playing back to the beginning position if in the backwards half of a forward/backward loop.

The following structure describes a loop:

playMode	**Rev. Word**	The type of looping to be performed. 0 = no looping 1 = Forward looping 2 = Forward/Backward looping If 0 is specified, the loop points are ignored during playback.
beginLoop	**Rev. Word**	A Marker ID of the marker to the begin position.
endLoop	**Rev. Word**	A Marker ID of the marker to the end position. The begin position must be less than the end position. If this is not the

case, the loop segment has zero or negative length and no looping occurs.

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

ASIF Note: The Apple IIGS Sampled Instrument Format also defines a chunk with ID of "`INST`," which is not the same as the Audio IFF Instrument Chunk. A good way to tell the two chunks apart in generic IFF-style readers is by the ckSize fields. The Audio IFF Instrument Chunk's ckSize field is always 20, whereas the Apple IIGS Sampled Instrument Format Instrument Chunk's ckSize field, for structural reasons, can never be 20.

The 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 the block as well. As more instruments come to 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.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`MIDI`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
MIDIdata	**Unsigned Bytes**	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.

The Audio Recording Chunk

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

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`AESD`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. For the Audio Recording Chunk, this value is always 24.
AESChannelStatusData	**24 Bytes**	These 24 bytes 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. This 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. Bits 2, 3, and 4 of byte zero are of general interest as they describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a `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.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`APPL`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. For the Audio Recording Chunk, this value is always 24.
OSType	**4 Bytes**	Identifies a particular application. For Apple II applications, these four bytes should always be '`pdos`' ($70 $64 $6F $73). In this case, the beginning of the data area is defined to be a Pascal string containing the name of the application. For Macintosh applications, this is simply the four-character signature as registered with Developer Technical Support.
AppSignature	**String**	Pascal string identifying the application.
data	**Bytes**	Data specific to the application.

Note: AppSignature does not exist unless OSType is "`pdos`." In all other cases, the data area starts immediately following the OSType field.

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

The Comments Chunk

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

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`COMT`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
numComments	**Rev. Unsigned Word**	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.
Comment	**Comment**	The comments. There are numComments of them.

The format of a comment is described below:

timeStamp	**Rev. Unsigned Long**	Indicates when the comment was created. Units are the number of seconds since 12:00 a.m. (midnight), January 1, 1904. This is the standard Macintosh time format. Macintosh routines to manipulate this time stamp may be found in *Inside Macintosh*, Volume II.

Note: Apple IIGS System Software 5.0.3 and later contains a Miscellaneous Tools routine, `ConvSeconds`, which can convert times in the format of timeStamp into standard ProDOS, GS/OS or HyperCard IIGS dates.

marker	**Rev. Word**	A Marker ID. If this comment is linked to a marker (to store a long description of a marker as a comment, for example), this is the ID of that marker. Otherwise marker is zero, indicating there is no such link.
count	**Rev. Word**	Count of the number of characters in the following text. By using a word instead of a byte, much larger comments may be created.
text	**Bytes**	The comment itself. If the text is an odd number of bytes in length, it must be padded with a zero byte to ensure that it is an even number of bytes in length. If the pad byte is present, it is not included in count.

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

The Text Chunks

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.

The Name Chunk

This chunk names the sampled sound.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`NAME`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
Name	**Bytes**	ASCII characters ($20–$7F) representing the name. There should be ckSize characters.

No more than one Name Chunk may exist within a `FORM AIFF`.

The Author Chunk

This chunk can be used to identify the creator of a sampled sound.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`AUTH`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.

author	**Bytes**	ASCII characters ($20–$7F) representing the name of the author of the sampled sound. There should be ckSize characters.

No more than one Author Chunk may exist within a `FORM AIFF`.

The Copyright Chunk

The Copyright Chunk contains a copyright notice for the sound. The copyright contains a date followed by the copyright owner. The chunk ID "`(c) `" serves as the copyright character (©). For example, a Copyright Chunk containing the text "1989 Apple Computer, Inc." means "© 1989 Apple Computer, Inc."

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`(c) `".
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk.
notice	**Bytes**	ASCII characters ($20–$7F) representing a copyright notice for the voice or collection of voices. There should be ckSize characters.

No more than one Copyright Chunk may exist within a `FORM AIFF`.

The Annotation Chunk

Use of this comment is discouraged within `FORM AIFF`. The more powerful Comments Chunk should be used instead.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`ANNO`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. You may think of this value as the offset to the end of the chunk. Note that this is a Reverse Long; the bytes are stored high byte first.
author	**Bytes**	ASCII characters ($20–$7F) representing the name of the author of the voices or collection of voices. There should be ckSize characters.

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. This precedence is illustrated in Figure 8.

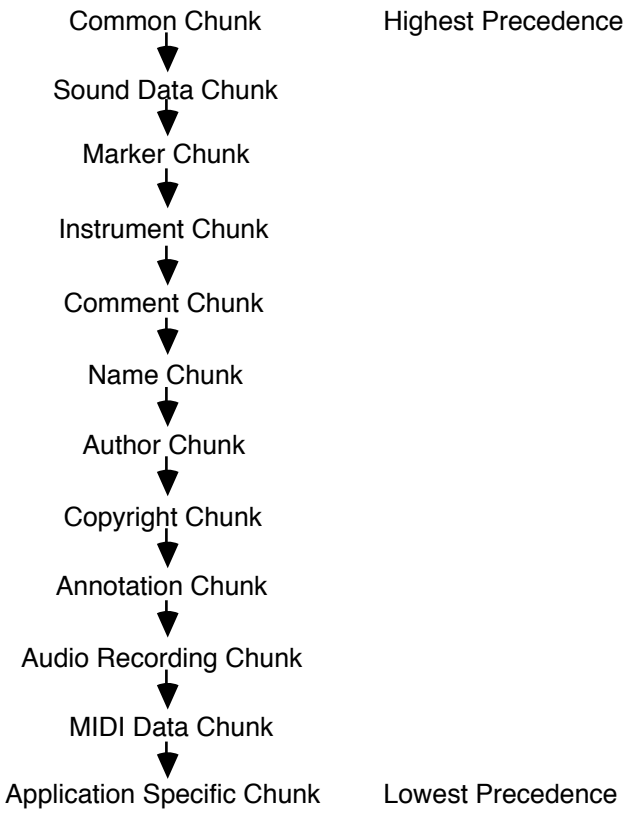

Figure 8–Chunk Precedence

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.

Figure 9 (on the following page) illustrates an example of a `FORM AIFF`. An Audio IFF file is simple a file containing a single `FORM AIFF`. The `FORM AIFF` is stored in the data fork of file systems that can handle resource forks.

Further Reference

- *Apple Numerics Manual*, Second Edition
- File Type Note File Type $D8, Auxiliary Type $0002, Apple IIGS Sampled Instrument Format
- *Audio Interchange File Format v1.3* (APDA)

- *AES Recommended Practice for Digital Audio Engineering—Serial Transmission Format for Linearly Represented Digital Audio Data*, Audio Engineering Society, 60 East 42nd Street, New York, NY 10165
- *MIDI: Musical Instrument Digital Interface, Specification 1.0*, the International MIDI Association.
- *"EA IFF 85" Standard for Interchange Format Files* (Electronic Arts)
- *"8SVX" IFF 8-bit Sampled Voice* (Electronic Arts)

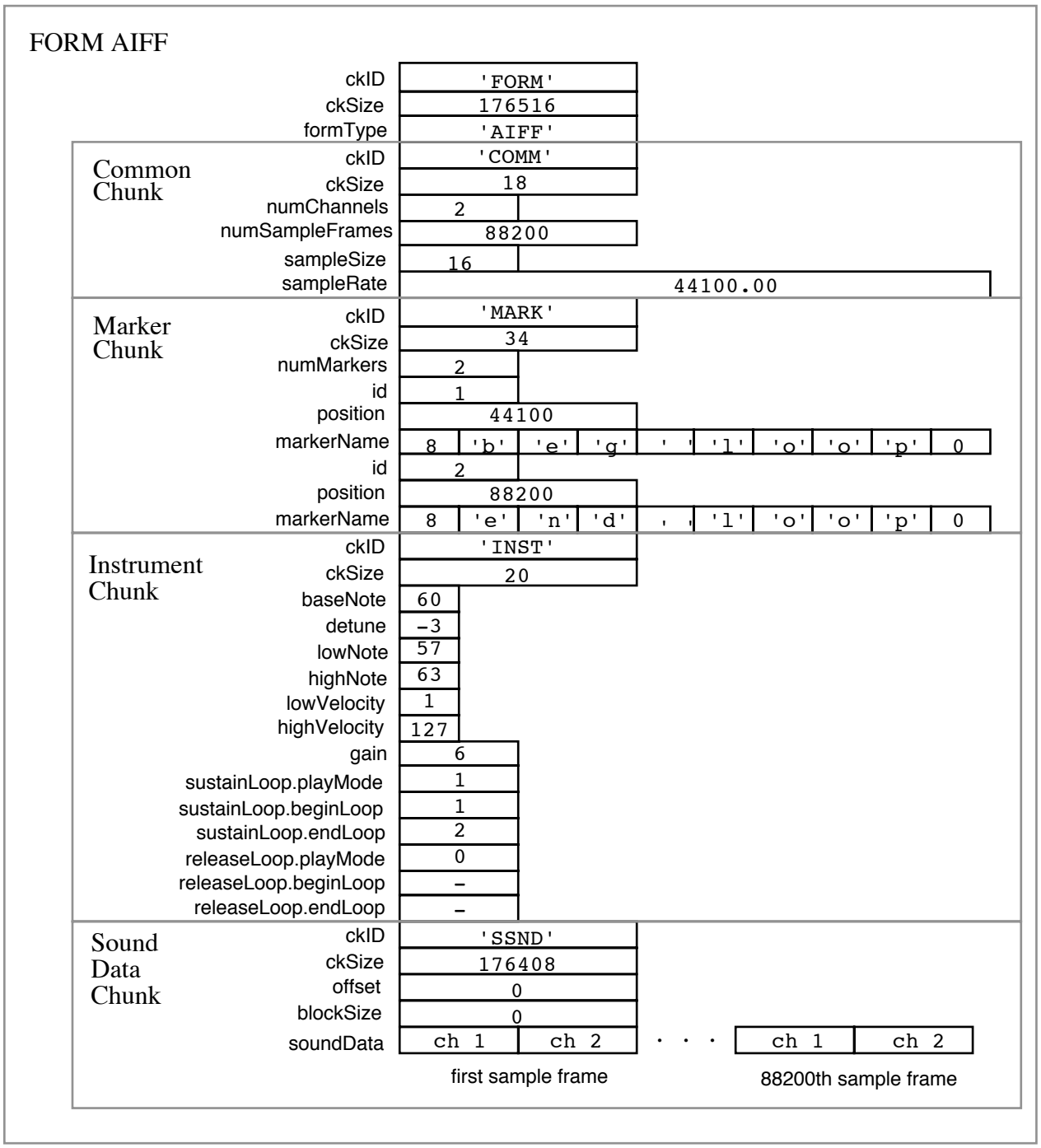

Figure 9–Sample FORM AIFF