Apple II
File Type Notes

Developer Technical Support

File Type: $D8 (216)
Auxiliary Type: $0001

Full Name: Audio Interchange File Format AIFF-C File
Short Name: AIFF-C File

Written by: Matt Deatherage March 1991

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

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 1.0 (February 3, 1991) of AIFF-C. This Note describes AIFF-C as it pertains to Apple II developers.

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

Although AIFF-C is an **interchange** format, 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. Apple Computer officially recommends that all sound applications read and write AIFF-C files.

AIFF-C is the result of several meetings held with music developers over a period of ten months during 1987 and 1988 as well as revisions to the original Audio IFF standard conducted during the summer of 1990. 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. AIFF-C 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 AIFF-C. 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 or AIFF-C.

Differences between Audio IFF and AIFF-C

The differences between the original AIFF and AIFF-C were kept to a minimum. Applications which currently support AIFF should be easily upgradable to AIFF-C. If you are unfamiliar with Audio IFF, you might want to skip directly to "The Chunk Concept" later in this Note.

The following changes have been made from AIFF:

- The `FORM` identifier was changed from "`AIFF`" to "`AIFC`". This distinguishes AIFF-C files from AIFF files. Existing AIFF programs, until they are upgraded, will simply ignore AIFF-C files. See the explanation below for this change.
- The Common Chunk has been extended to include a compression type ID and a compression type name. AIFF-C is thus capable of storing compressed audio data generated from any compression algorithm.
- The Sound Data Chunk can contain compressed audio data. The Chunk format has not been modified.
- The Sound Accelerator (Saxel) Chunk is new. It is designed to eliminate initial "artifacts" caused by the decompression algorithm when playback begins at a random point defined by a Marker.
- The Format Version Chunk is new. This Chunk is designed to provide a smooth transition for potential future upgrades to the AIFF-C specification.

Transition from FORM AIFF to FORM AIFC

Renaming the `FORM` type from `AIFF` to `AIFC` was done to minimize confusion for the end user. Imagine a possible scenario if the FORM type was **not** changed: A user running an application which stored compressed audio in AIFF-C format would save his compressed audio data as an AIFF File type (via the "Save As ..." dialog box). He could then launch another application which reads AIFF, but not AIFF-C, and the application would not be able to play his sound. The application may even crash. By making the difference between the file types explicit, the user will not experience this problem. The user still won't be able to transfer compressed audio data to the second application, but at least he will know why.

Here are the guidelines which developers should follow to aid the transition from AIFF to AIFF-C:

1. Applications which currently *read* `FORM AIFF` files should also be able to read FORM AIFC files.
2. Applications which currently *create* `FORM AIFF` files should maintain this capability for now, but should offer the `FORM AIFC` format as the default option to the user.
3. New applications which have not supported AIFF should strongly consider supporting only AIFF-C, at least for the *creation* of audio files.

The Chunk Concept

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

AIFF-C, 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 AIFF-C is 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 AIFF-C 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 AIFF-C 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 AIFF-C 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 AIFF-C files, formType is always "`AIFC`." This indicates that the chunks within the `FORM` pertain to sampled sound. A `FORM` chunk of formType AIFC is called a `FORM AIFC`.

chunks	**Bytes**	The chunks contained within the `FORM`. These chunks are called local chunks. A `FORM AIFC` along with its local chunks make up an AIFF-C file.

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

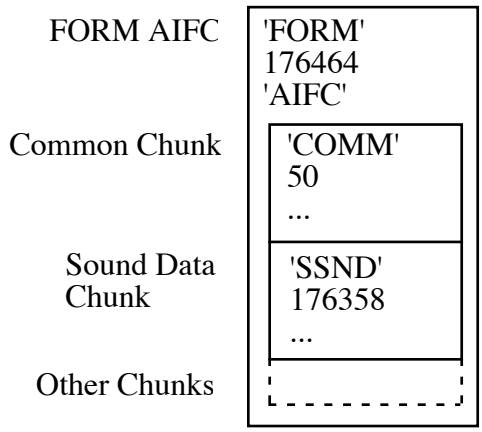

Figure 1–Simple AIFF-C File

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

The `FORM AIFC` is stored in a file with file type $D8 and auxiliary type $0001. AIFF-C files may be identified in other file systems as well. On a Macintosh under MFS or HFS, the `FORM AIFC` is stored in the data fork of a file with file type "AIFC." This is the same as the formType of the `FORM AIFC`.

Note: Applications should not store any data in the resource fork of an AIFF-C 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 AIFF-C file names have the extension ".AFC."

A more detailed visual example of an AIFF-C 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 AIFC` 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 AIFC` must be able to read the required chunks and can choose to selectively ignore the optional chunks. A program that copies a `FORM`

`AIFC` should copy all the chunks in the `FORM AIFC`, even if it chooses not to interpret the optional chunks.

Dealing with Unrecognized Local Chunks

When reading an IFF file, your program may encounter local chunk types that it doesn't recognize, perhaps extensions defined after your program was written. In a `FORM AIFC`, this situation also applies to Application-Specific Chunks with unrecognized application signatures. (The application signature acts as a chunk subtype.) Clearly your program cannot process the contents of unrecognized chunks.

So what should your program do when it encounters unrecognized chunks in an IFF `FORM`? The safest thing is to simply discard them while reading the `FORM`. If your program copies the `FORM` without edits, then it's nicer (but not necessary) to copy unrecognized chunks also. But if your program modifies the data in any way, then it **must** discard all unrecognized chunks since it can't possibly update the unrecognized data to be consistent with the modifications.

To ensure that this standard remains usable by all developers across machine families, only Apple Computer, Inc. should define new chunk types for `FORM AIFC`. 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 "AIFF-C Suggestions."

The Format Version Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`FVER`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID. For the Format Version Chunk, this is always 4.
timeStamp	**Rev. Unsigned Long**	Indicates the version of AIFF-C with which this file was created. Units are the number of seconds since 12:00 a.m. (midnight), January 1, 1904. This document describes Version 1 of AIFF-C, for which the value of timeStamp should be $A2805140 (May 23, 1990, 2:40 PM). Only Apple may alter the value of timeStamp.

Do not confuse the Format Version with the creation date of the file. The Format Version refers to the rules embodied in this, or future, documents which specify how an AIFF-C file is arranged. When your application checks for compatibility with the Format Version chunk, do not do a range check (e.g. less than or equal to this date). You must do an exact comparison of dates to know for certain that your application can correctly read and process a specific AIFF-C file. Do **not** modify the timeStamp value. If you have a request for a new Format Version, please submit it to Apple Computer at the address in "About File Type Notes". Through this mechanism where only Apple Computer can issue official AIFF-C releases with new timeStamps, we can ensure the maximum compatibility of AIFF-C files across applications.

The Format Version Chunk is **required**. One and only one Format Version Chunk must appear in a `FORM AIFC`.

Why the Format Version Chunk was added

"Gee, if we'd had a Version Chunk in AIFF, we wouldn't have to change the `FORM` type for AIFF-C."—*an anonymous AIFF File Type Note author (circa 1990)*

From the above proverb, we gained the wisdom to include a Format Version Chunk in the AIFF-C specification. The philosophy is that the Chunk names which you recognize will contain information in the format you are familiar with. If you don't find a Chunk which your application requires, then examine the Format Version Chunk to determine if the file is corrupted or if there is a mismatch between your application and the file. In any case, you'll be able to give a more enlightened message to the user.

See how the following steps simplify your life (and ours) to determine if a `FORM AIFC` is usable:

When reading an AIFF-C file:

1. First, find the `FORM AIFC` field. If you don't find it, issue an alert like "This file doesn't contain an AIFF-C standard audio recording.", then exit from these directions.

2. Try to find all the chunks which are critical to your application (probably `COMM` and `SSND`, but we can imagine an application that only needs the `COMM` chunk, e.g. to determine the playback duration).

   If found, those familiar chunk IDs indicate that the chunk contents are in the format you expect. You're golden. Exit these directions.

3. If not found, don't crash yet. Instead, check for the Format Version Chunk.

   If you can find it and it does not contain a date which you recognize, issue an alert like "This file contains an unrecognized version of the AIFF-C standard." You may also want to indicate the file's Format Version and the format versions which your application recognizes.

   Otherwise, issue an alert with text similar to "This file seems to be chopped cabbage." It might also be nice to say which Chunks are missing.

Remember:

- In order to survive interchange and format evolution, reader programs must be robust about chunk order, missing chunks, and unexpected chunks.

- Contrary to the original Audio IFF specification, when a program encounters an unrecognized chunk, it should just skip it. Do **not** copy it to a new, *edited* file. This is the general rule in IFF because there's no way to maintain the integrity of unrecognized chunks when the surrounding data is edited.

How the Format Version Chunk will help potential future upgrades

If and when we design evolutionary changes to the file format, we will try to make the new representation backward compatible (e.g. just add new chunk types). If we must change the format of existing data, then we will change the relevant Chunk IDs. For example, let's say that the `INST` Chunk needs to be upgraded to have more than 2 loop points. In this case, we would replace the `INST` Chunk with a new Chunk—let's call it "`LOOP`". In the transition time between widespread adoption of the new `LOOP` Chunk, a `FORM` could contain both the old `INST` Chunk and the new `LOOP` Chunk. Applications which know about the new `LOOP` Chunk would be able to process it correctly, while preserving the `INST` Chunk for other applications. Applications which do not use the `INST` or `LOOP` Chunks are unaffected. Applications which need the old `INST` Chunk can still use it, but should upgrade to the new `LOOP` Chunk since there is no longer any guarantee that other (editing) applications will preserve the old `INST` Chunk.

Here's how we would have upgraded AIFF to handle compressed audio, if we had had a Format Version Chunk already in AIFF:

- Compression is optional. What follows is *only* for the compressed case.
- Don't change the format of the `COMM` Chunk. Existing programs can still read it.
- Add a "Compression Descriptor" Chunk containing the 4-letter compression type code and the compression name string. (The code is for programs. The string is for alerts when the code is unrecognized.)
- Replace the `SSND` Chunk with a Compressed Sound-Data Chunk `"CSND"`. (Existing programs will ignore it.)
- Change the Format Version date (for the sake of alerts).
- Add the optional Saxel Chunk.

We chose to change the `FORM` type from `AIFF` to `AIFC` because, lacking the Format Version Chunk, existing applications would not be able to issue a helpful error message. Some existing applications may even crash if they did not find the `SSND` Chunk.

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 22 plus the length of the string (the string includes a pad byte when needed to make it an even number of bytes in length).
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.
compressionType	**4 Bytes**	The ID for the compression algorithm used. A table of possible values is in the "More About Compression" section of this Note.
compressionName	**String**	A Pascal string containing a human-readable description of the compression algorithm used. A table of possible values is in the "More About Compression" section of this Note.

One, and only one, Common Chunk is required in every `FORM AIFC`.

Sample Points and Sample Frames

A large part of interpreting AIFF-C files revolves around the two concepts of sample points and sample frames.

An uncompressed 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.

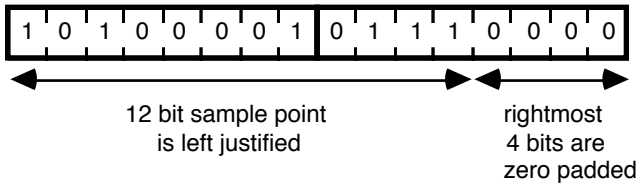

Figure 2–A 12-Bit Sample Point

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.

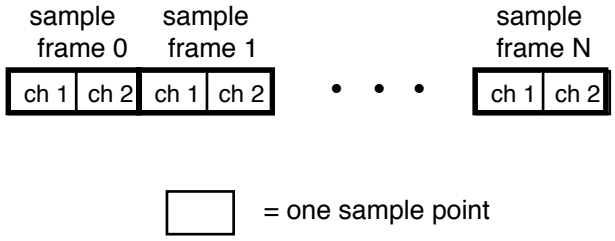

Figure 3–Sample Frames for Multichannel Sound

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.

For compressed sounds, these definitions essentially hold but are modified slightly. See the "More About Compression" section later in this Note for all the details.

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. If the sound data is padded to make it an even number of bytes in length, the pad byte is not included in ckSize.
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 AIFC`.

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.

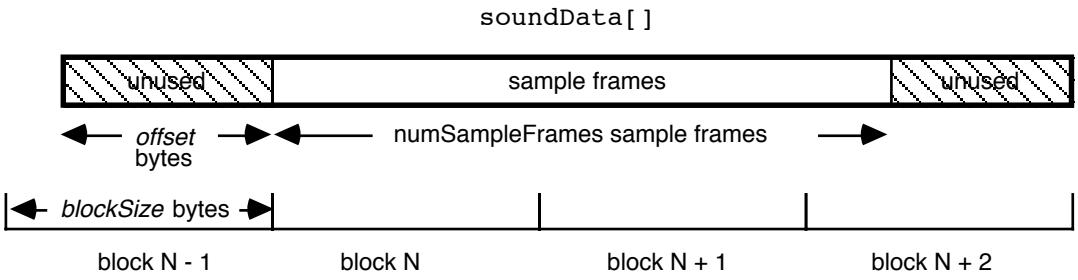

Figure 5–Block-Aligned Sound Data

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 AIFF-C files. Applications that write block-aligned sound data should set blockSize to the appropriate block size. Applications that modify an existing AIFF-C 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 AIFC`. The number can be any positive, non-zero integer, as long as no other marker within the same `FORM AIFC` 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. If the length of the string is not an even number of bytes, include a zero pad byte.

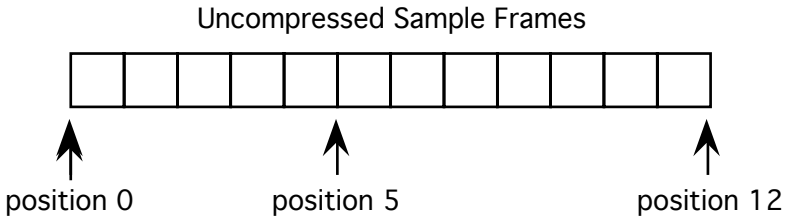

Figure 6–Uncompressed Sample Frame Marker Positions

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

Apple recommends that audio editor programs update markers when the audio data is edited.

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

Important: If a segment of sound data containing one or more Markers is relocated in the sound stream, the Markers within the segment being moved must be recalculated. If a segment of sound data is being deleted, all Markers within that segment should be deleted and all Markers after that segment must be adjusted. If sound data is inserted at a point in the sound data stream, all Markers after that point must be adjusted. Any Saxels (see "Sound Accelerator Chunks" later in is Note) which are associated with the updated or deleted Markers must also be updated if affected by the new Marker values. Updating Markers in some cases may have implications in the user interface and the application designer should consider when the user should be notified or asked about the consequences of an edit.

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.

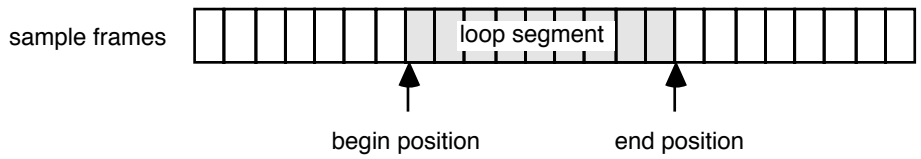

Figure 7–Sample Frame Looping

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.

When looping on **compressed** sound data, make sure to pay particular attention to setting the markers to the **expanded** sound data (see the section on the Marker Chunk in this Note). Extra attention may be required for smooth playback between the end of the looped data and the beginning of the looped data due to discontinuities in the sound data encountered by the expansion algorithm. In this case, the best recourse may be to modify sound samples in the beginning or end part of the loop, or to avoid compressing the looped data.

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

ASIF Note: The Apple IIGS Sampled Instrument Format also defines a chunk with ID of "`INST`," which is not the same as the AIFF-C Instrument Chunk. A good way to tell the two chunks apart in generic IFF-style readers is by the ckSize fields. The AIFF-C 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.

More About Compression

AIFF-C supports multiple types of compression. Although AIFF-C itself has no intrinsic knowledge of any compression scheme (that is, there is nothing in any of the chunks that depends on the behavior of any compression scheme), it does include flexible support for multiple compression types—for identifying the compression type, for informing users about it, and for using structures like markers and loops with compressed sound data.

The Common Chunk contains a four-byte compressionType and an ASCII compressionName, identifying the compression used to both the program and to the user. AIFF-C debuts with five pre-defined compression values:

compressionType	ASCII compressionName	meaning
NONE	not compressed	uncompressed, that is, straight digitized samples
ACE2	ACE 2-to-1	2-to-1 IIGS ACE (Audio Compression / Expansion)
ACE8	ACE 8-to-3	8-to-3 IIGS ACE (Audio Compression / Expansion)
MAC3	MACE 3-to-1	3-to-1 Macintosh Audio Compression / Expansion
MAC6	MACE 6-to-1	6-to-1 Macintosh Audio Compression / Expansion

Table 1—Existing Compression Schemes

Apple IIGS programs will normally not have access to MACE decompression, and the reverse is true for Macintosh program with ACE decompression. In these cases, you can use the ASCII compressionName to inform the user of the sound's compressed source, giving him significantly more information than "I can't decompress this sound."

The compression of sound is fairly straightforward. The diagrams below should help illustrate the point for the four standard Apple compression algorithms. Examples are for 8-bit linear monophonic sound samples.

Original uncompressed single channel sound data:

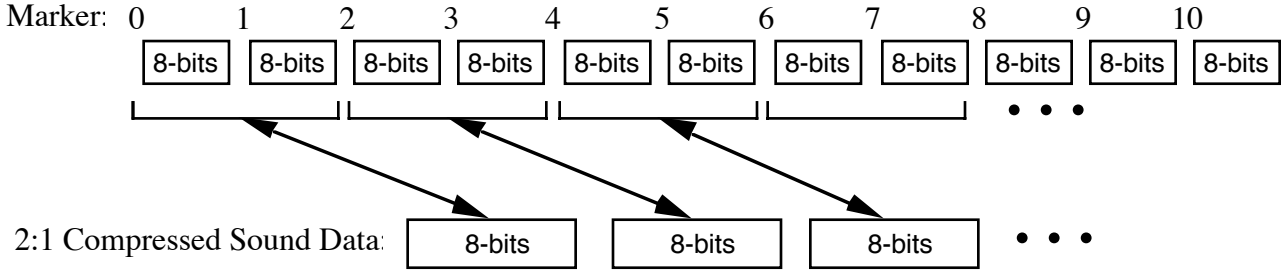

Figure 8—ACE 2:1 Apple IIGS Tool Frame size = 1 byte

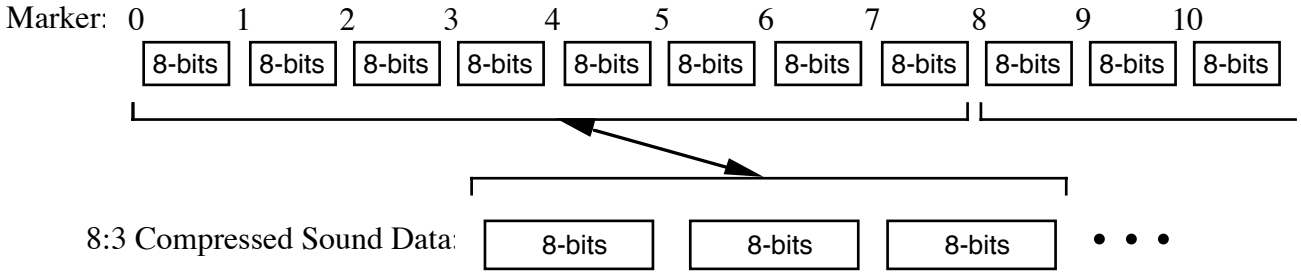

Figure 9—ACE 8:3 Apple IIGS Tool Frame size = 3 bytes

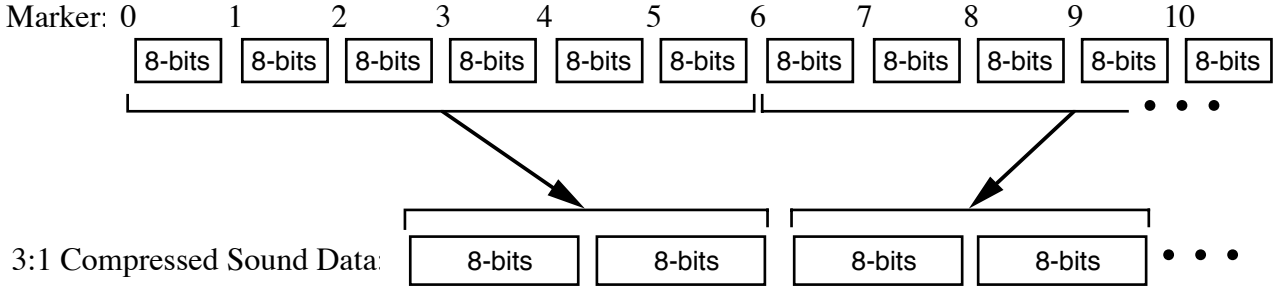

Figure 10—3:1 Macintosh Audio Compression & Expansion
Frame size = 2 bytes

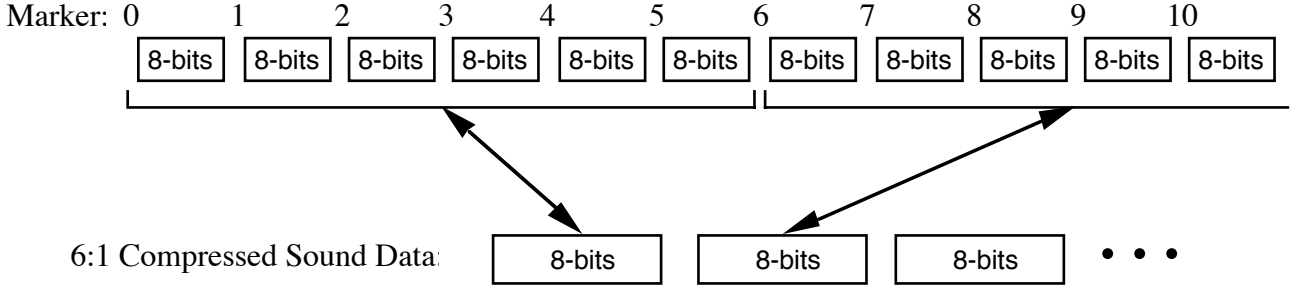

Figure 11—6:1 Macintosh Audio Compression & Expansion
Frame size = 1 byte

The sample frame size is the basic unit of a compressed data block. For the Macintosh 6:1 and the ACE 2:1 utilities, the frame size is 1 byte. For the Macintosh 3:1 utility, the frame size is 2 bytes. For the ACE 8:3 utility, the frame size is 3 bytes.

For storage of multichannel compressed sounds, the conventions listed in the Common Chunk section should be followed, using a sample frame of compressed sound data in place of uncompressed samples. Here are some examples:

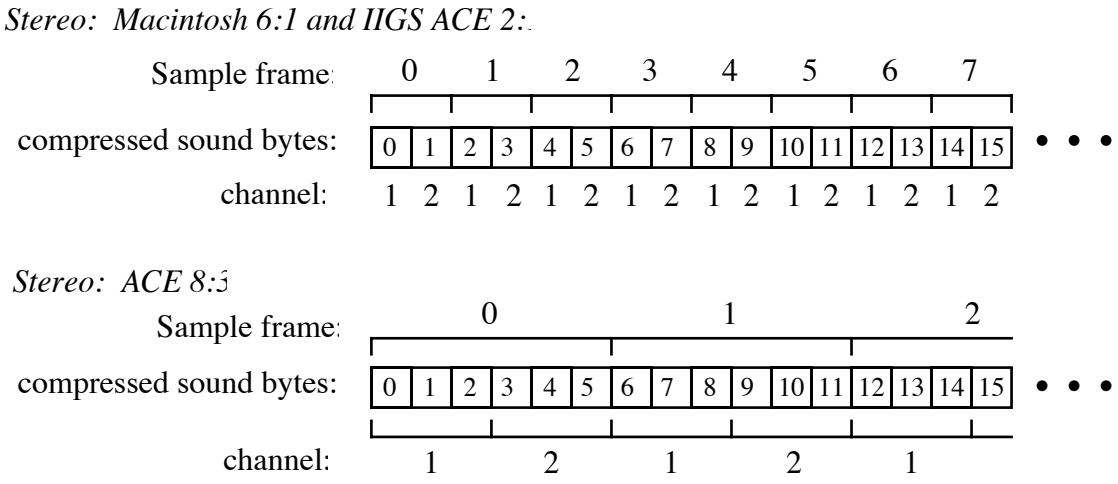

Figure 12—Multichannel Compressed Sounds

Markers

Markers positions (see Marker Chunk section) are targeted to expanded (uncompressed) sound data. Thus, a calculation must be done to map from a position in the compressed data stream to the target position in the uncompressed sound data. Fortunately, the existing compression utilities are linear - there is a straight multiplicative ratio between the size of compressed sound data to uncompressed sound data. Here is a table which can help you to calculate the actual Marker position, given an offset index into the compressed (single channel) sound data:

Compression		(Bytes)				Single channel sound data			
Macintosh 3:1	Compressed data offset:	0	2	4	6	8	10	12	14
	Marker Position:	0	6	12	18	24	30	36	42
Macintosh 6:1	Compressed data offset:	0	1	2	3	4	5	6	7
	Marker Position:	0	6	12	18	24	30	36	42
ACE 2:1	Compressed data offset:	0	1	2	3	4	5	6	7
	Marker Position:	0	2	4	6	8	10	12	14
ACE 8:3	Compressed data offset:	0	3	6	9	12	15	18	21
	Marker Position:	0	8	16	24	32	40	48	56

Table 2—Marker Positions in Compressed Sound

It is possible to specify a Marker position which is not a multiple of the compression rate (e.g. Marker position of 19 for Macintosh 3:1 compressed sound data). In this case, the playback system must be contain enough intelligence to:

1. Expand a compressed sample frame and discard the initial expanded sample(s) before playback, and
2. Stop playback of samples before the last expanded sample. In the case of a compressed sound which must be looped, this capability provides added accuracy in determining the best loop points.

The Sound Accelerator Chunk

Audio decompression algorithms contain internal parameters which track the behavior the sound being expanded. As these internal parameters depend on the history of the previous sound samples, a simple attempt to begin playback at arbitrary positions in the compressed sound data would result in artifacts and distortion of the initial portion of the expanded sound. A Saxel stores information about the compressed sound at a Marker position, thus providing a means for high quality playback of random selections of compressed sound data.

Background

Generally, a decompressor must start from the beginning of the compressed data stream. It requires running state (e.g. internal filter parameters or recently decompressed samples) to decompress the next sample. To start playback at a marker point somewhere within the audio stream, you could:

a. decompress the data from the beginning and start playing once you reach the marker, or
b. use additional data to locate the marked point within the compressed data stream and load up the decompressor state, then start playing, or
c. compute the marked point within the compressed data stream (only possible for fixed-ratio compression types), initialize the decompressor as if it were starting at the beginning, and ignore the startup transient (only useful for decompressors that would "settle down" in this case).

Method a is always possible as a fall-back. Method b is much faster, if you have the required data. And that's what Saxel (Sound Accelerator) chunks are for. Method c may be acceptable for certain applications and/or certain classes of audio compression.

A Sound Accelerator (Saxel) chunk is used in combination with a Marker when the sound data is compressed. The saxel carries the required data to locate a point in the compressed data stream and to initialize the decompressor. Saxels enable method b and a modified method a:

d. decompress the data from the previous marker that has a Saxel and start playing once you reach the desired marker.

Note: As of this writing, the Saxel formats for Macintosh Audio Compression and Expansion (MACE) compression had not been finalized, and are therefore not included in this Note. Apple would welcome any ideas you have on the topic at the address in "About File Type Notes."

The Sound Accelerator (Saxel) Chunk has the following format:

ckID	**4 Bytes**	The ID for this chunk. These four bytes must be "`SAXL`."
ckSize	**Rev. Long**	The length of this chunk, excluding ckSize and cdID.
numSaxels	**Rev. Word**	The number of saxels in this chunk. Multiple Saxel Chunks are allowed in a single `FORM AIFC`. Since the total amount of Saxel data for a heavily-edited sound file may be quite large, it

		may be easier for applications to store the various Saxels independently of each other.
Saxels	**Saxels**	The Saxels themselves.

A Saxel has the following format:

MarkerID	**Rev. Word**	The ID of the marker for which the sound accelerator data is to be used. It's considered good practice to supply a Saxel for every marker—that way, you don't have to guess which markers will be used as playback points.
size	**Rev. Word**	The length in bytes of the sound accelerator data, saxelData. The data must be padded with a zero byte at the end if necessary to make it an even number of bytes in length. This pad byte, if present, is not included in size.
saxelData	**Bytes**	The Sound Acceleration Data

Sound Acceleration Data for Apple IIGS ACE Compression

Beginning with System Software 5.0.3, ACE has a way to retrieve the exact state of the decompression algorithm, expressly for the inclusion of such data in Saxel Chunks of AIFF-C. Future versions of ACE will include a tool call to retrieve this data. For version 1.2 of ACE, the values to include as saxelData are the first 16 bytes of ACE's direct page. ACE's direct page can be obtained from the Tool Locator routine `GetWAP` with `systemOrUser` = $0000 and `tsNum` = 29.

For versions later than 1.2 of ACE, the information can be obtained through the new ACE calls `GetACEExpState` ($0D1D) and `SetACEExpState` ($0E1D). Both calls take the same parameter list and return no errors:

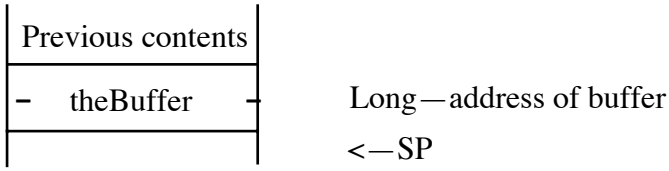

Figure 13—Stack Diagram for new ACE calls

Versions of ACE prior to 1.2 do not support retrieving the compression state and should not be used with AIFF-C.

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 AIFF-C 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. If the length of the MIDI data is odd, a pad byte must follow it to make it even.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a `FORM AIFC`. If MIDI System Exclusive messages for several instruments are to be stored in a `FORM AIFC`, 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 AIFC`.

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). For non-Apple applications, these four bytes should be '`stoc`'. In these cases, 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`" or "`stoc`". In all other cases, the data area starts immediately following the OSType field.

Be sure to plan for the future when defining the data section of your Application Specific Chunk. Use a version numbering scheme or other appropriate method that will enable the current and future versions of your applications to interpret the data in the `FORM AIFC`. Specifically, the current application should be able to inform the user when a new version is encountered which it cannot handle (and possibly even prompt the user with a solution). Future applications should be able to handle older versions of the data or guide the user to a solution.

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

The Comments Chunk

The Comments Chunk is used to store comments in the `FORM AIFC`. "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 AIFC`.

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 AIFC`.

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 AIFC`.

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 AIFC`.

The Annotation Chunk

Use of this comment is discouraged within `FORM AIFC`. 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 AIFC`.

Chunk Precedence

Several of the local chunks for `FORM AIFC` 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 listed in Table 2.

Format Version Chunk	*Highest precedence*
Common Chunk	
Instrument Chunk	
Saxel Chunk	
Comments Chunk	
Marker Chunk	
Sound Data Chunk	
Name Chunk	
Author Chunk	
Copyright Chunk	
Annotation Chunk(s)	—in the order they appear in the FORM
Audio Recording Chunk	
MIDI Data Chunk(s)	
Application Specific Chunks	*Lowest precedence*

Table 2–Chunk Precedence

The Format Version 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.

AIFF-C Examples

Illustrated below are examples of several `FORM AIFC` files. An AIFF-C file is simply a file containing a single `FORM AIFC`.

These examples have been designed to illustrate several of the possible variations of sound data and Chunk formats you may encounter. A careful study of these examples will clarify the Chunk specifications. Remember that the Chunks may appear in *any* order in a `FORM AIFC`—the order shown here is only for the sake of the examples.

Example 1—A file containing approximately 4.476 seconds of 8-bit monophonic uncompressed sound data sampled at 22.25454 KHz:

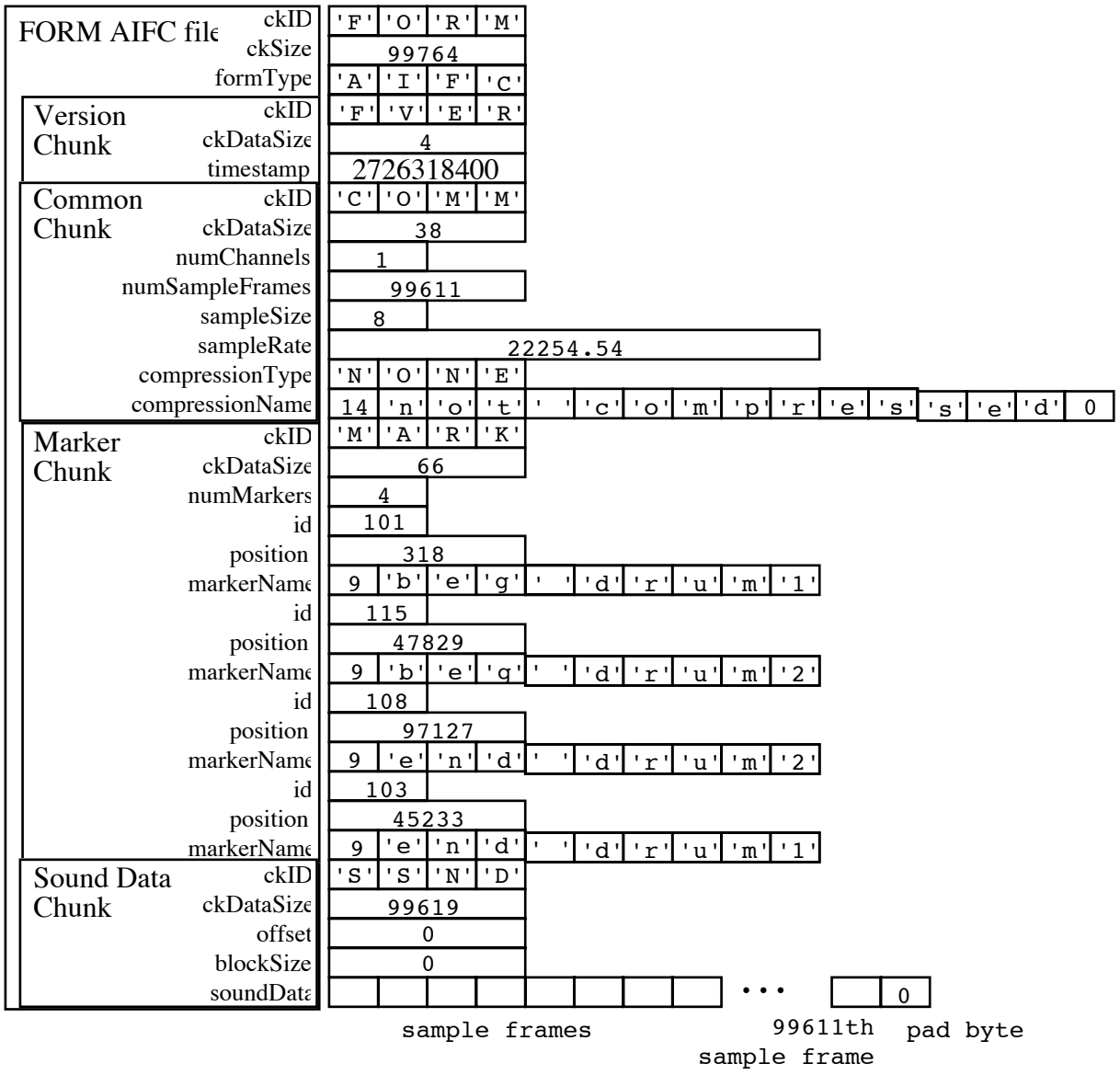

Figure 14–Sample FORM AIFC #1

Example 2—A file containing approximately 28.972 seconds of 8-bit sound data sampled at 22.25454 and compressed by a factor of 2 using the Apple IIGS Audio Compression and Expansion tool.

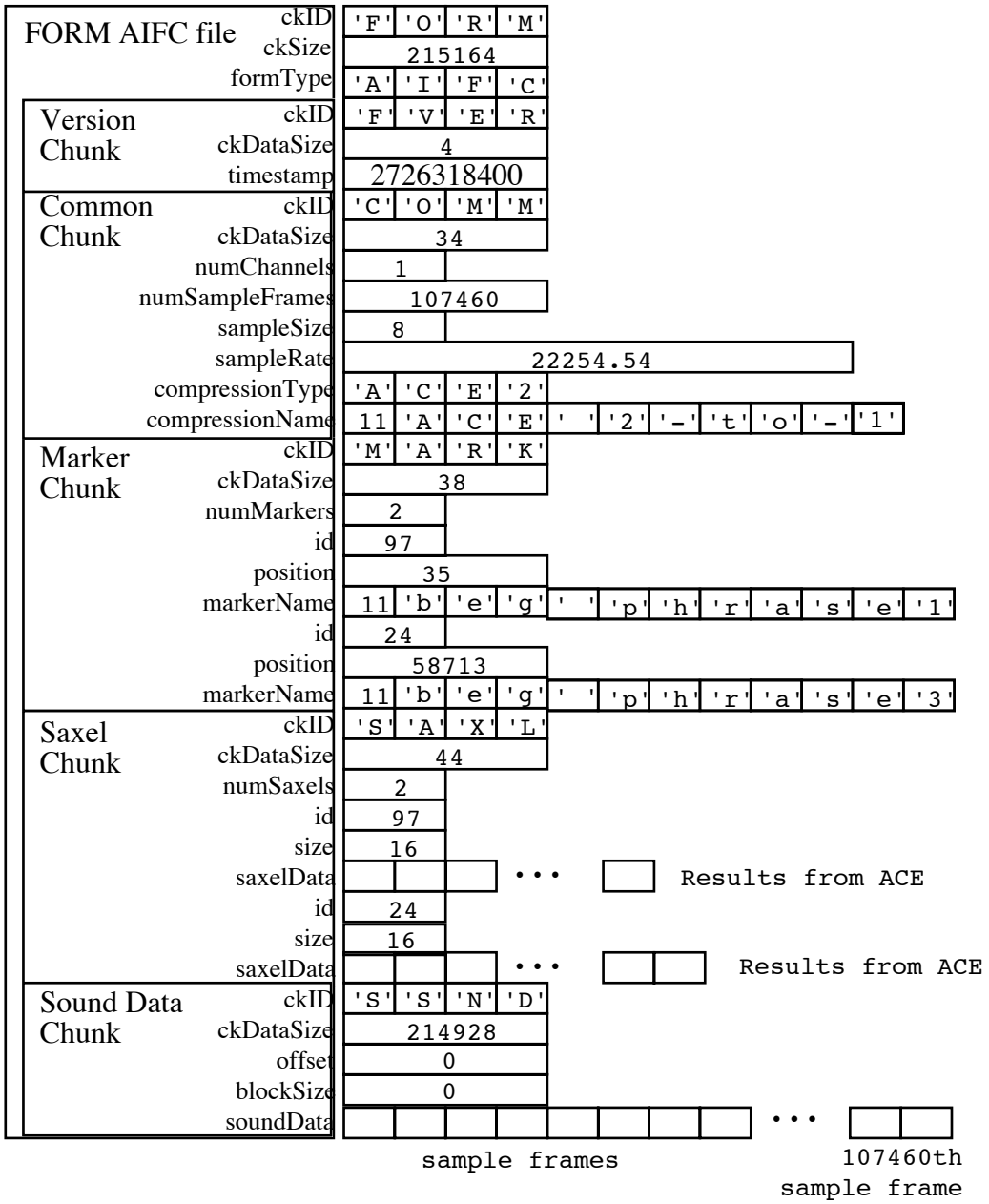

Figure 15–Sample FORM AIFC #2

Example 3—A file containing approximately 2.325 seconds of 16-bit stereo uncompressed sound data sampled at 44.1 KHz (CD quality).

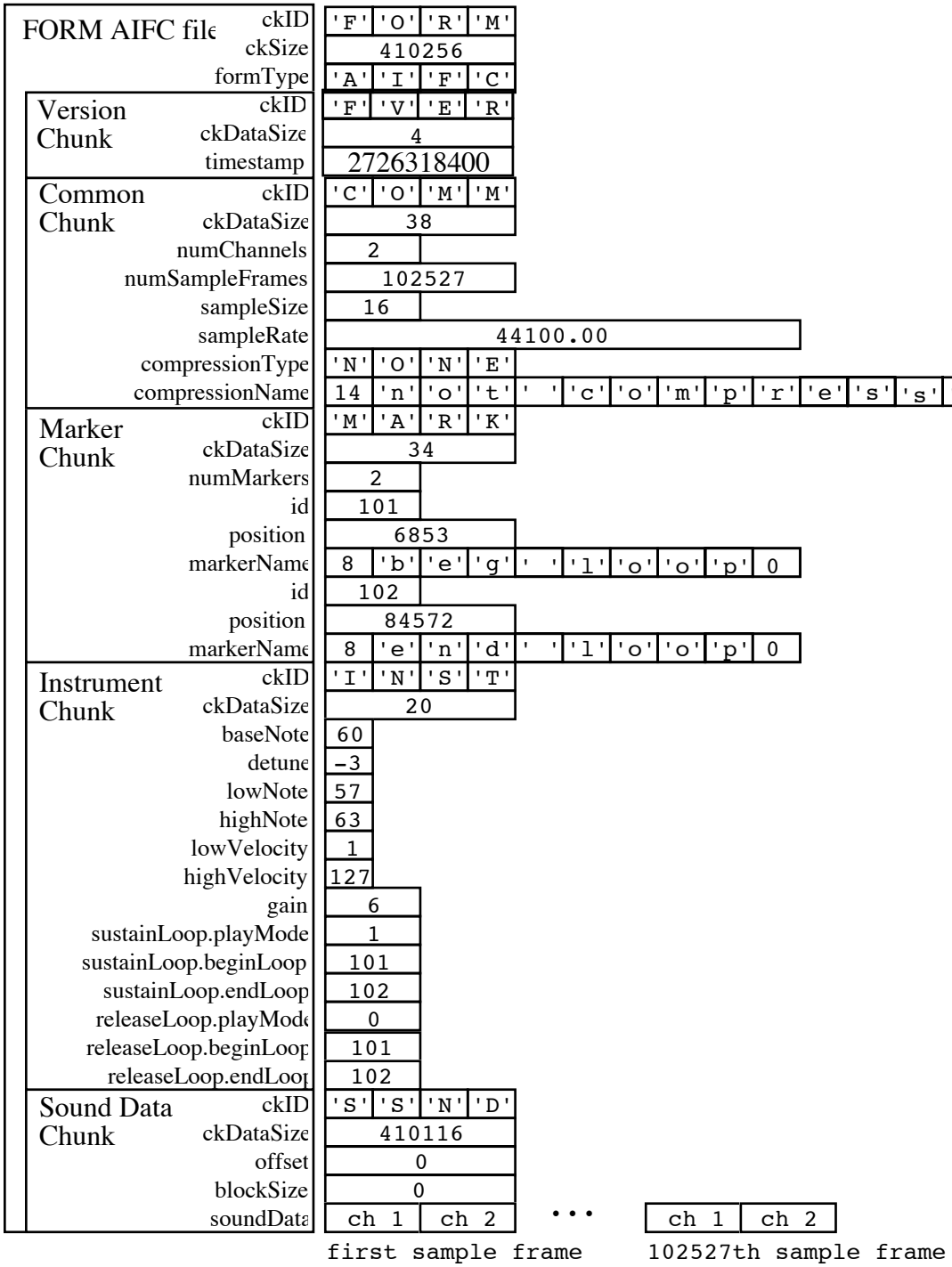

Figure 16–Sample FORM AIFC #3

Further Reference

- *Apple Numerics Manual*, Second Edition
- File Type Note File Type $D8, Auxiliary Type $0000, Audio Interchange File Format
- 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)