*** DRAFT ***

Draft date: 8/26/91

Audio Interchange File Format AIFF-C

A revision to include compressed audio data

Apple Computer, Inc.

DRAFT NOTES

This is the same draft as the 9/30/90 document. Although it is still listed as "draft," a number of users including Apple Computer have been using this format. Recently, we have assembled a number of changes to the document that are in process, and therefore we will continue calling this a draft copy until those changes are in place at which time an official release can be made. It is important to note that the changes are expected to be informational only, and do not change the definitions. They may, however, affect recommended practices. For example, we will be providing a code example as to how to convert the extended precision numbers used to specify frequency, and also may provide recommeded practice as to conversion from floats to integer and back in relation to audio useage. Since this is now an "external document," the confidentiality notices in the 9/30/90 draft have been removed.

Why a new format?

AIFF-C is being defined because *AIFF does not allow for compressed audio data*. AIFF-C adds the ability to store compressed audio data in a standard manner. Naturally, AIFF-C also allows the storage of uncompressed audio data. The "C" in AIFF-C signifies its extension to handle *compressed* audio data.

Differences between AIFF 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.

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 algorithms.
- 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 de-compression algorithms 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. Let's examine what would happen otherwise: 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 would then run another application which reads AIFF, but not AIFF-C, and the application would not be able to play his sound, or it may even crash. By making explicit the difference between the file types, the user would 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.

Pearls of Wisdom

From experience gathered over the period of time since the original AIFF has been released, we offer the following advice to those developers who are beginning to implement either AIFF or AIFF-C. For the purposes of this section only, we will refer to both AIFF and AIFF-C as simply AIFF.

Chunk Ordering

Remember that there is no order imposed on the Chunks! They may appear in any order in an AIFF file. It may seem logical to place the Common Chunk at the beginning, followed by the other Chunks and terminated with the large Sound Data Chunk, but this is not a requirement. Your application which reads an AIFF file should be designed to get a Chunk, identify it, then process it with no supposition as to which Chunk it is until it has been identified.

Modifying Chunks

If your application allows modification of a Chunk, you must also take on the responsibility of updating other Chunks which are based on the modified Chunk. An example is cutting some sound data from the Sound Data Chunk - if there are Markers which pointed to data which was removed, those Markers should also be deleted (some user interaction may be appropriate in certain cases). Also, other Markers may need to be re-calculated to preserve their position to the correct (shifted) sample frames.

BEWARE: If your application allowed an AIFF file to be edited (modified) and if there are Chunks in the file which you do not recognize, you **must** discard those unknown Chunks when you save the file!!! (This is a modification of the guideline stated in the previous AIFF specification.) Another application which uses those unrecognized Chunks could be seriously affected if those Chunks depend on the Chunks which you modified. Of course, if your application is simply copying the AIFF file without modifications, then it should also copy the unrecognized Chunks. We recommend that your application understand each and every Chunk which is listed in this specification to handle this situation in the best possible way.

Registering New Compression Types

You must register your compression type with Apple to establish an official *compressionType* and *compressionName*. You should also describe the format and usage of the Sound Accelerator Chunk for your compression type. By registering this information, other developers will know about and be able to include your compressed sound format in their applications. This will allow end users the ability to transfer compressed audio data between applications - which is the goal of this specification. Please see appendix B on how to contact Apple to register your compression type.

Number of Sample Frames

The clearest indication of the number of sample frames contained within the file is obtained from the *numSampleFrames* parameter in the Common Chunk and **not** the *ckDataSize* parameter in the Sound Data Chunk. The *ckDataSize* parameter in the Sound Data Chunk is padded to include the fields which follow it in addition to the actual sound data but does not include the zero pad byte at the end if the total number of sound data bytes is odd. Simple!

Remember the Pad Byte!

Each Chunk must contain an *even* number of bytes. For those Chunks whose total contents would yield an odd number of bytes, a zero pad byte must be added at the end of the Chunk. This pad byte is **not** included in *ckDataSize*, which indicates the size of the data in the Chunk. It helps to keep this in mind as you seek through an AIFF file to get to the next Chunk.

Format Version Chunk

The section entitled *"When reading an AIFF-C file"* in the Format Version Chunk is of special importance.

BEWARE: Obsolete format version of AIFF-C data in existence

A CD-ROM has been released to developers entitled the "Macintosh System Softwre 7.0 - May 1990 Alpha Development Release". A file named "JLG MacWorld" exists in the *MultiTrack* folder inside the *Goodies* folder. The "JLG MacWorld" file was created with a draft version of the AIFF-C specification and it is **NOT compatible** with the most recent AIFF-C specification. Its FORM type is 'AIFS' (as opposed to the correct 'AIFC') and it does not contain a Format Version Chunk. Your applications should NOT recognize this file as a valid AIFF-C file. Do not use this file as a compatibility test for your AIFF-C applications.

Table of Contents

Audio Interchange File Format AIFF-C

A Standard File Format for Audio Data
Apple Computer, Inc.
Draft: July 30, 1990

1.0 INTRODUCTION

The Audio Interchange File Format AIFF-C provides a standard for storing uncompressed or compressed sampled sounds. The format can store monaural or multichannel sampled sounds in a range of sample rates and sample widths. The format is extensible to handle new compression types and application-specific data.

AIFF-C is based on Audio IFF (AIFF) which conforms to the "*EA IFF 85" Standard for Interchange Format Files* developed by Electronic Arts.

AIFF-C is designed for interchange, although application designers should find it flexible enough to use as an everyday data storage format as well. If an application uses a different storage format, it can convert to and from the AIFF-C format defined here. This will facilitate the sharing of sound data between applications and across various computer platforms.

Data types

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

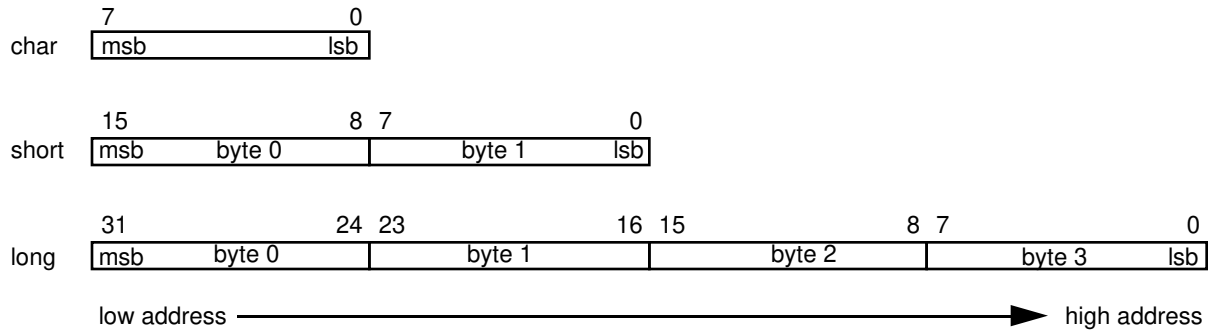
`char`:	8 bits, signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).
`unsigned char`:	8 bits, unsigned. Contains any number from zero to 255 (inclusive).
`short`:	16 bits, signed. Contains any number from -32,768 to 32,767 (inclusive).
`unsigned 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, one byte count followed by text bytes followed—when needed—by one pad byte. The total number of bytes in a pstring must be even. The pad byte is included when the number of text bytes is even, so the total of text bytes + one count byte + one pad byte will be even. 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. Upper/lower case is significant, that is, IDs are compared using a simple 32-bit equality check.
`OSType`:	32 bits. A concatenation of four characters, as defined in *Inside Macintosh, vol II.* Upper/lower case is significant, that is, OSTypes are compared using a simple 32-bit equality check.

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. Numbers are stored high-byte first, as follows:



Referring to Audio Interchange File Format AIFF-C

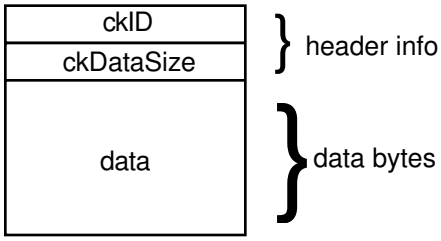
The official name for this standard is *Audio Interchange File Format AIFF-C.* If an application needs to present the name of this format to a user, such as in a "Save as…" dialog box, the name can be abbreviated to *AIFF-C* or *Audio IFF-C*.

2.0 FILE STRUCTURE

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 document recaps those portions of "EA IFF 85" that are germane to AIFF-C. For a more complete discussion of "EA IFF 85", please refer to the documents *"EA IFF 85" Standard for Interchange Format Files* and *A Quick Introduction to IFF*.

An "EA IFF 85" file is built up from 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.



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

```
typedef struct {

       ID              ckID;           /* chunk ID                       */
       long            ckDataSize;     /* chunk data size, in bytes      */

       char            ckData[];       /* data                           */

} Chunk;
```

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

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

ckData is 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 *ckDataSize*.

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

```
typedef struct {

       ID              ckID;           /*  'FORM'  */
       long            ckDataSize;

       ID              formType;       /*  'AIFC'  */
       Chunk           chunks[];

} FormAIFCChunk;
```

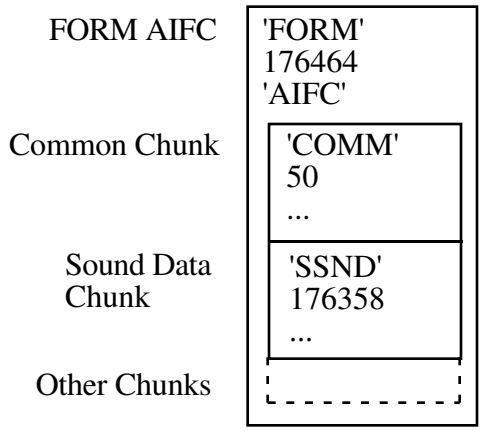
ckID is always 'FORM'. This indicates that this is a FORM chunk.

ckDataSize 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, much like a Mac file type. For AIFF-C files, *formType* is 'AIFC'. This indicates that the chunks within the FORM pertain to sampled sound according to this AIFF-C standard. A FORM chunk of *formType* 'AIFC' is called a *FORM AIFC*.

chunks are the chunks contained within the FORM. These chunks are called *local chunks* since their own *ckID*'s are local to (i.e. specific to) FORM AIFC. A FORM AIFC along with its local chunks make up an AIFF-C file.

Here's an example of a simple AIFF-C file. It consists of a file containing single FORM AIFC Chunk which contains two local chunks, a Common Chunk and a Sound Data Chunk. (Please refer to Appendix A for more detailed examples.)



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

On an Apple II, the FORM AIFC 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 AIFC is stored in the data fork of an AIFF-C file. The Macintosh file type of an AIFF-C file is 'AIFC'. This is the same as the *formType* of the FORM AIFC.

Macintosh or Apple II applications should not store any information in the resource fork of an AIFF-C file, as this information might not be maintained by other AIFF-C editors. Applications can use *Application Specific Chunks*, 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 AIFF-C file names have a ".AFC" extension.

Local Chunk Types

The formats and *ckID*s of the local chunk types found within a FORM AIFC are described in the following sections.

The Common Chunk is required in a FORM AIFC. If the sampled sound has greater than zero length, then the Sound Data chunk is required. 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.

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, too. But if your program modifies the data in any way, then it **must** discard all unrecognized chunks. That's because it can't possibly update the unrecognized data to be consistent with the modifications.

To insure that this standard remains usable by everyone, Apple Computer, Inc. will act as the central repository of new chunk types for FORM AIFC. If you have suggestions for new chunk types, Apple is happy to listen! Please refer to Appendix B for instructions on how to send comments to Apple.

3.0 FORMAT VERSION CHUNK

The Format Version Chunk contains a date field to indicate the format rules for an AIFF-C specification. This will enable smoother future upgrades to this specification.

Format Version Chunk

The format for the data within a Format Version Chunk is shown below.

```
#define AIFCVersion1     0xA2805140      /*  Version 1 of AIFF-C              */
                                         /*   this is 2726318400  in decimal  */

typedef struct {

      ID              ckID ;          /*  'FVER'          */
      long            ckDataSize ;    /*   4              */
      unsigned long   timestamp ;     /*  AIFCVersion1    */

} FormatVersionChunk;
```

ckID is always 'FVER'.

ckDataSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckDataSize*. For this Chunk, *ckDataSize* has a value of 4.

timeStamp indicates when the format version for the AIFF-C file 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.

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 - see appendix B on how to contact Apple. 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 had had a Version Chunk in AIFF, we wouldn't have had to change the FORM type for AIFF-C." - *Anonymous (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 AIFC 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 app 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 AIFC standard." You may also want to indicate the file's format version and the format versions which your application recognizes.

   Otherwise, issue an alert like "This file seems to be malformed." Maybe 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 AIFF spec, 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 to a new name. 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, 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.

4.0 COMMON CHUNK

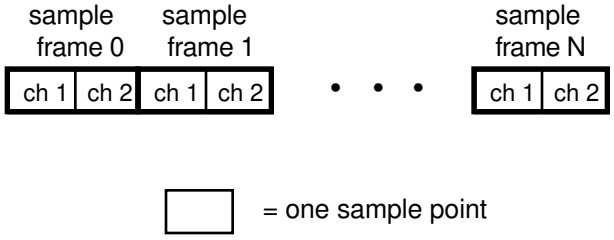
The Common Chunk describes fundamental parameters of the sampled sound.

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

typedef struct {

       ID             ckID;               /*  'COMM'  */
       long           ckDataSize;

       short          numChannels;        /* # audio channels */
       unsigned long  numSampleFrames;    /* # sample frames = samples/channel */
       short          sampleSize;         /* # bits/sample */
       extended       sampleRate;         /* sample_frames/sec */
       ID             compressionType;    /* compression type ID code */
       pstring        compressionName;    /* human-readable compression type name */

} CommonChunk;
```

ckID is always 'COMM'. *ckDataSize* is the size in bytes of the data portion of the chunk. It does not include the 8 bytes used by *ckID* and *ckDataSize*. For the Common Chunk, *ckDataSize* is 22 + the size of the pstring. (The pstring includes a pad byte when needed to fill out to an even number of bytes.)

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.* For uncompressed sound data, the total number of sample points in the file is *numSampleFrames * numChannels*.

sampleSize is the number of bits in each sample point of uncompressed sound data. 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.* For compressed sound data, *sampleSize* indicates the number of bits in the original sound data before compression.

sampleRate is the sample rate at which the sound is to be played back, in *sample frames* per second.

compressionType is used by programs to identify the compression algorithm, if any, used on the sound data. *compressionName* is used by people to identify the compression algorithm. Use *compressionType* to select the decompression routine. Use *compressionName* to display a human-readable message when it you don't have the needed decompression routine. Remember to pad the end of *compressionName* with a zero byte if the pstring length is not an even number of bytes, but do not include the pad byte in the count.

The initial values are:

compressionType	English *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

Note: *compressionType* is a standard 32-bit ID value that identifies the compression algorithm. In contrast, the *compressionName*'s value can be country-specific, e.g. stored in French or Spanish.

Compression types are allocated by Apple. Other third party compression schemes are welcome, but you must reserve the compressionType ID with Apple. Please see Appendix B.

The Apple IIGS ACE (Audio Compression / Expansion) and the Macintosh Audio Compression / Expansion encoding schemes are documented in appendix C.

One and only one Common Chunk must appear in every FORM AIFC.

5.0 SOUND DATA CHUNK

The Sound Data Chunk contains the actual sample frames.

```
#define     SoundDataID    'SSND'          /* ckID for Sound Data Chunk */

typedef struct {

       ID              ckID;           /*  'SSND'  */
       long            ckDataSize;

       unsigned long   offset;
       unsigned long   blockSize;
       char            soundData[];

} SoundDataChunk;
```

ckID is always 'SSND'.

ckDataSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckDataSize.* It does include the 8 bytes taken by *offset* and *blockSize*. If *soundData[]* contains an odd number of bytes, a pad byte with a value of zero is added at the end to preserve an even length for this Chunk. This pad byte, if present is **not** included in *ckDataSize*. To avoid confusion, the actual number of sample frames should always be obtained from the *numSampleFrames* parameter in the Common Chunk.

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. See also *Block-Aligning Sound Data*, below.

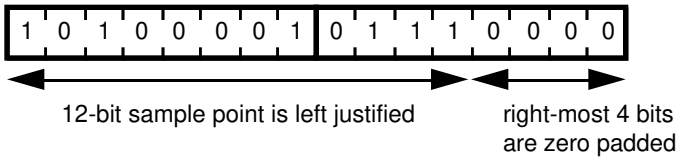
soundData contains the sample frames that make up the sound. The number of sample frames in the *soundData* is determined by the *numSampleFrames* parameter in the *Common Chunk*. If *soundData[]* contains an odd number of bytes, a zero pad byte is added at the end (but not used for playback).

Linear Sound Data (not compressed)

Each sample point in a sample frame is a linear, 2's complement value. Sample points are from 1 to 32 bits wide, as determined by the *sampleSize* parameter in the *Common Chunk*.

Each sample point is 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 (using a shift-left instruction), with the remaining bits zeroed. The remaining low-order bits at the right end are set to zero.

As an example, the 12-bit sample, binary 101000010111, is stored left justified in two bytes:



Sample Frames

The sample points within a sample frame are packed together as described in the section on the *Common Chunk*, above. Sample frames are stored contiguously in order of increasing time. There are no pad bytes between samples or between sample frames.

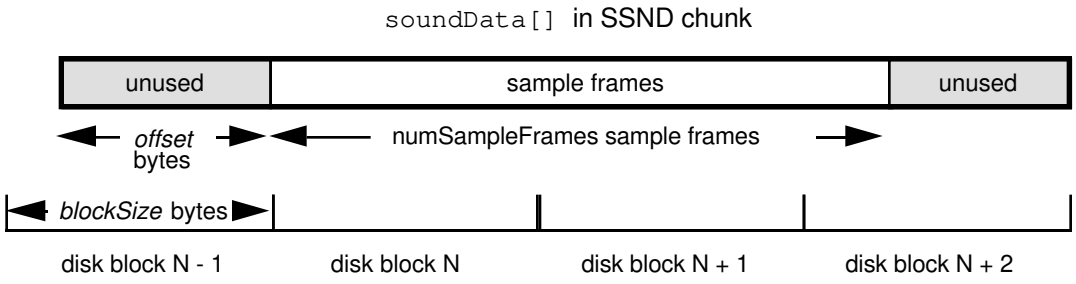
Compressed Sound Data

The *soundData* is compressed according to the *compressionType* parameter in the Common Chunk.

Appendix C describes the encoding format for the existing Apple Computer audio compression utilities and the use of Marker and Saxel Chunks (see below) with the various compression types. Developers wishing to include their own compression type should contact Apple (see appendix B), and provide documentation similar to that contained in appendix C. Some applications may not need to understand the encoding format of compressed audio information and may only need to copy the sound data for interchange purposes. Applications which desire to manipulate the compressed sound data, such as an editing application, will need to observe the encoding schemes used by the compressed sound types they wish to edit.

Block-Aligning Sound Data

There may be some applications that, to enable real time recording and playback of audio, wish to align the sampled sound data on a fixed-size disk block. 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 disk block N. This is accomplished by skipping the first *offset* bytes of *soundData*. The *soundData* array may also extend beyond valid sample frames in order to end on a disk block boundary.

blockSize specifies the size in bytes of the alignment block. 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 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 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 AIFC.

6.0 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.

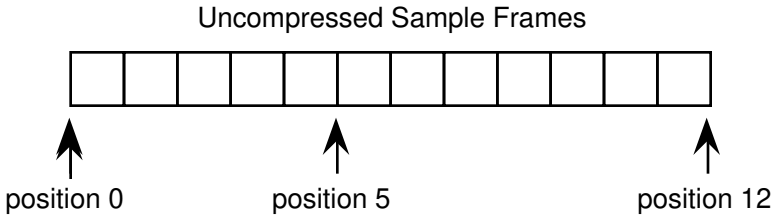
Markers

A marker has the following format.

```
typedef     short     MarkerId;

typedef struct {

      MarkerId            id;            /*  must be > 0  */
      unsigned long       position;      /*  sample frame number  */
      pstring             markerName;

} Marker;
```

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

The marker's position in the sound data is indicated 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.



For *compressed* sound data, the marker's position is based on *expanded* (uncompressed) sound data, and not the position of the compressed sample frame. This allows fine-grained resolution for placing marker points exactly where they are needed (especially important for loop points). A single byte of compressed sound data may expand into many bytes of expanded sound data, preventing high resolution of markers based on compressed data. The mapping of compressed sound data sample frames to expanded sound data sample frames is easily done for the existing Apple audio compression algorithms. These mappings are described in appendix C.

We recommend that audio editor programs update the markers when the audio data is edited.

markerName is a pstring containing the name of the mark. Remember to include a pad byte when needed to round out a pstring to an even number of bytes.

Note: Some "EA IFF 85" files store C-style strings (text bytes followed by a null terminating character) instead of `pstrings`. AIFF-C uses `pstrings` because they are more efficiently skipped over when scanning through chunks. A program can skip over a `pstring` by adding the string count and the pad size 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;          /*  'MARK'  */
       long                 ckDataSize;

       unsigned short       numMarkers;
       Marker               markers[];

} MarkerChunk;
```

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

numMarkers is the number of markers in the Marker Chunk.

numMarkers, if non-zero, 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 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 re-calculated. 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 appendix D) 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.

7.0 COMMENTS CHUNK

The Comments Chunk is used to store comments about markers in the FORM AIFC. "EA IFF 85" has an *Annotation Chunk* that can be used for comments, but the Comments Chunk adds to each comment (1) a timestamp and (2) a reference 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;     /* comment creation date */
      MarkerId            marker;        /* comments for this marker number */
      unsigned short      count;         /* comment text string length */
      char                text[];        /* comment 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 annotations or 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 is the comment itself. This text must be padded with a byte at the end as needed to make it an even number of bytes long. This pad byte, if present, is not included in *count*.

Comments Chunk Format

```
#define     CommentID      'COMT'         /* ckID for Comments Chunk  */

typedef struct {

      ID                  ckID;          /*  'COMT'  */
      long                ckDataSize;

      unsigned short      numComments;
      MarkerComment       comments[];

} CommentsChunk;
```

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

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 AIFC.

8.0 SOUND ACCELERATOR (SAXEL) CHUNK - ** Under Construction! **

**** WARNING****

****This is a rough proposal and discussion only at this time!!****

The Saxel Chunk is designed to provide high-quality playback from any random point, indicated by a Marker, in the compressed audio data stream . There are several possible ways that this mechanism could be implemented and we have not come to any final conclusions yet. One possible method has been documented in appendix D for your review and comments. Please send us your feedback on the Saxel Chunk proposal or on any other method you would recommend.

The need for a Saxel Chunk arises from the behavior of audio de-compressors which, for the most part, rely on some history of the recently de-compressed samples to predict the value for the next sample to be de-compressed. De-compressing the audio stream beginning at a random point would cause initial audio artifacts to be heard before the algorithm's internal de-compression parameters stabilized.

Please refer to appendix D for the Sound Accelerator Chunk proposal.

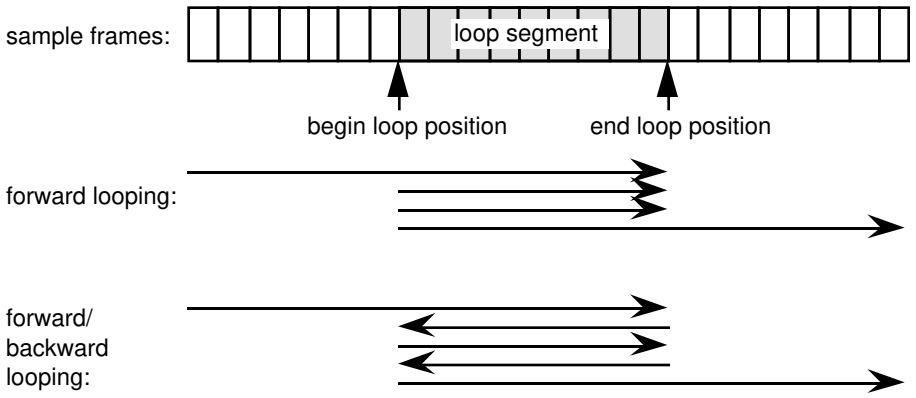
9. INSTRUMENT CHUNK

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

Looping

A portion of the sound data can be repeated in order to lengthen the sound. This portion, called the *loop segment*, is repeated until interrupted by something like the release of a key on a sampling keyboard.

There are two ways to play a loop: forward looping and forward/backward (or "ping pong") looping.



To implement forward looping, play the loop segment over and over again. To implement forward/backward looping, play the loop segment forwards, then backwards, and repeat this forwards/backwards pair over and over again.

Forward/backward (or "ping pong") looping is nice because its automatically seamless. To do forward looping without clicks, carefully pick the loop begin and end positions so they match up without a seam.

If looping is being done on *compressed* sound data, make sure to pay particular attention to setting the markers to the *expanded* sound data (see the section on Markers). 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 structure below describes a loop:

```
typedef struct {

      short          playMode;
      MarkerId       beginLoop;
      MarkerId       endLoop;

} Loop;
```

playMode specifies which type of looping to perform:

```
#define     NoLooping               0
#define     ForwardLooping          1
#define     ForwardBackwardLooping  2
```

`NoLooping` means ignore these loop points during playback.

beginLoop and *endLoop* are marker ids that mark the begin and end positions of the loop segment. The begin position must be less than the end position so the loop segment will have a positive length. (If this is not the case, then ignore this loop segment. 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;          /*  'INST'  */
       long            ckDataSize;

       char            baseNote;
       char            detune;
       char            lowNote;
       char            highNote;
       char            lowVelocity;
       char            highVelocity;
       short           gain;
       Loop            sustainLoop;
       Loop            releaseLoop;

} InstrumentChunk;
```

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

baseNote is the pitch of the originally recorded sound. 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 is used to make small tuning adjustments to the sound in case it wasn't recorded exactly in tune. *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* suggest the useful playback range for this sound data. Use this sound data to play a note between the low and high notes, inclusive. (Look for some other sound data to play notes beyond this range. The *baseNote* does not have to be within this range.) Units for *lowNote* and *highNote* are MIDI note values.

gain is the amount 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. To play louder and softer notes, further adjust the playback gain.

lowVelocity and *highVelocity* suggest the useful note-on velocity (volume) range for this sound data. Use this sound data to play a note between *lowVelocity* and *highVelocity*, inclusive. (Look for some other sound data to play notes beyond this range.) Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

sustainLoop specifies a loop to play when an instrument is sustaining a sound.

releaseLoop specifies a loop to play 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.

[TBD] Extensions to store multiple samples in an AIFC file, e.g. one sample per octave.

> • *The [TBD] here is waiting to hold a decision on how to store multiple instruments in a FORM AIFC. We discussed the possibility that this could be done simply by adding two more markerIds to the InstrumentChunk—start and end markers, and by specifying exactly how to use this, including how to set the "baseNote" field so the different audio samples (perhaps one per octave) can have different sample rates.*

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

10. 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 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.

```
#define     MIDIDataID     'MIDI'          /* ckID for MIDI Data Chunk */

typedef struct {

       ID                  ckID;           /*  'MIDI'  */
       long                ckDataSize;

       unsigned char       MIDIdata[];

} MIDIDataChunk;
```

ckID is always '`MIDI`'.

ckDataSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckDataSize*. If *ckDataSize* is odd, a pad byte must follow this chunk.

MIDIData contains a stream of MIDI data.

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.

11. 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;          /*  'AESD'  */
       long                 ckDataSize;

       unsigned char        AESChannelStatusData[24];

} AudioRecordingChunk;
```

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

12. 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;                   /*  'APPL'  */
       long            ckDataSize;

       OSType          applicationSignature;
       char            data[];

} ApplicationSpecificChunk;
```

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

ckDataSizeize indicates the length in bytes of the data portion plus the length of the OSType field. The data must be padded with a byte at the end as needed to make it an even number of bytes long. This pad byte, if present, is not included in *ckDataSize*.

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.

For applications which run on other than Apple computers, the application signature should always be 'stoc'. 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.

data is the data specific to the application. The data must be padded with a byte at the end as needed to make it an even number of bytes long. The guidelines listed under *applicationSignature* for Apple II and non-Apple applications must also be followed for this data portion.

As a general guideline for developers using this Chunk, be sure to plan for the future when defining the structure of the *data* portion. 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 to a solution). Future applications should be able to handle older versions of 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.

13. TEXT CHUNKS - NAME, AUTHOR, COPYRIGHT, ANNOTATION

These four chunks are included in the definition of many IFF FORMs. 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                 ckDataSize;

       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 ID, note that the 'c' is lowercase and there is a space (0x20) after the close parenthesis.

ckDataSize is the size of the data portion of the chunk, in this case the number of characters in *text.*

text contains pure ASCII characters. It is neither a `pstring` nor a C string. The number of characters in *text* is determined by *ckDataSize*. The meaning of the *text* depend on the chunk type, 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 AIFC.

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 AIFC.

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 character '©'. 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 AIFC.

text contains a comment. Use of this chunk is discouraged within FORM AIFC. The more refined *Comments Chunk* should be used instead. The Annotation Chunk is optional. Any number of Annotation Chunks may exist within a FORM AIFC.

14. 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 a defined precedence for chunks:

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*

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 A. Examples of a FORM AIFC

Illustrated below are examples of several FORM AIFC files. An AIFF-C file is simply a file containing a single FORM AIFC. On a Macintosh, the FORM AIFC is stored in the data fork of a file and the file type is '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.

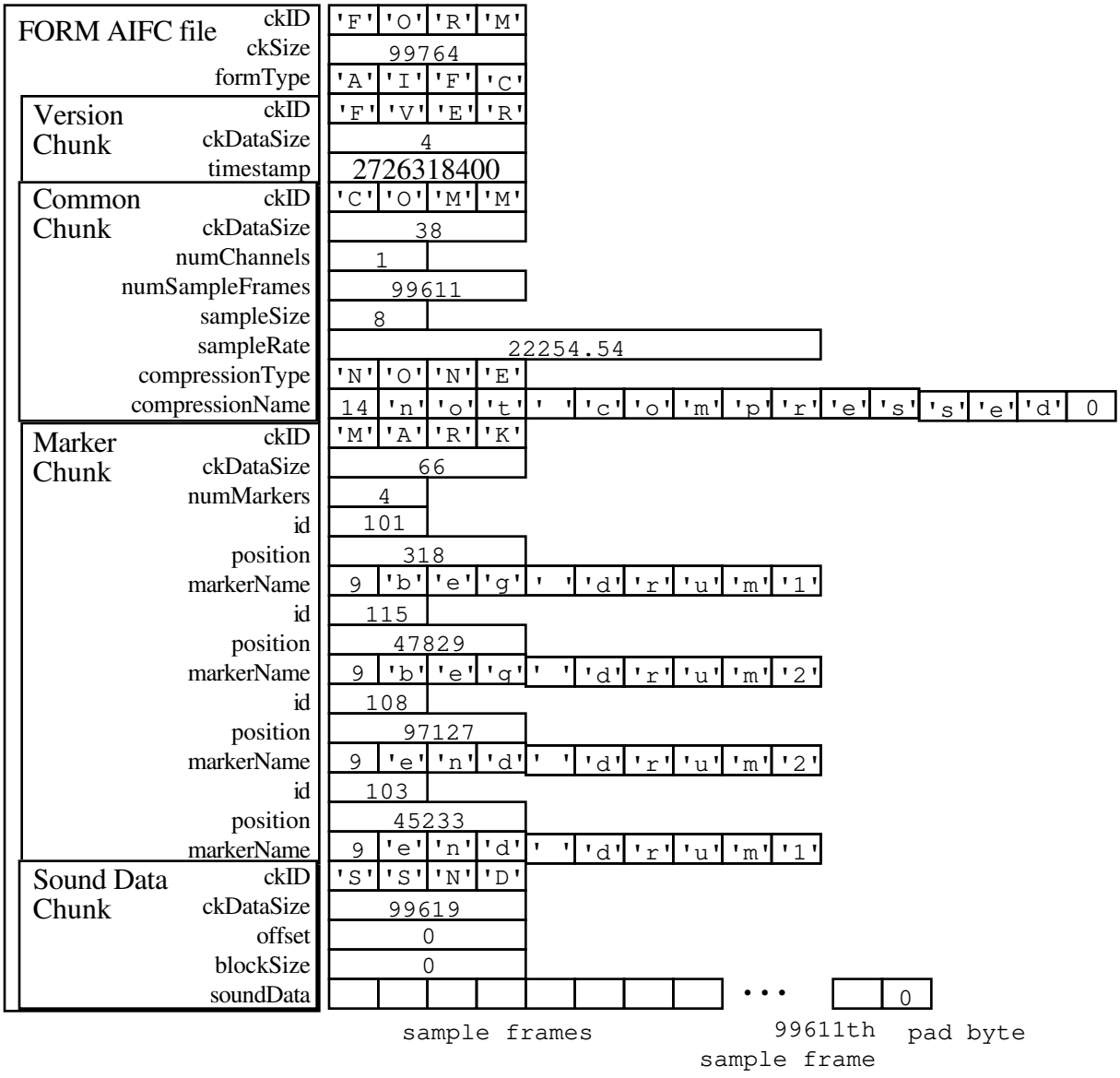
List of Examples:

1. 8-bit monophonic sound data sampled at 22.25454 kHz. The sound data is not compressed.

2. 8-bit monophonic sound data sampled at 22.25454 kHz and compressed by a factor of 3 using the Macintosh Audio Compression & Expansion utility.

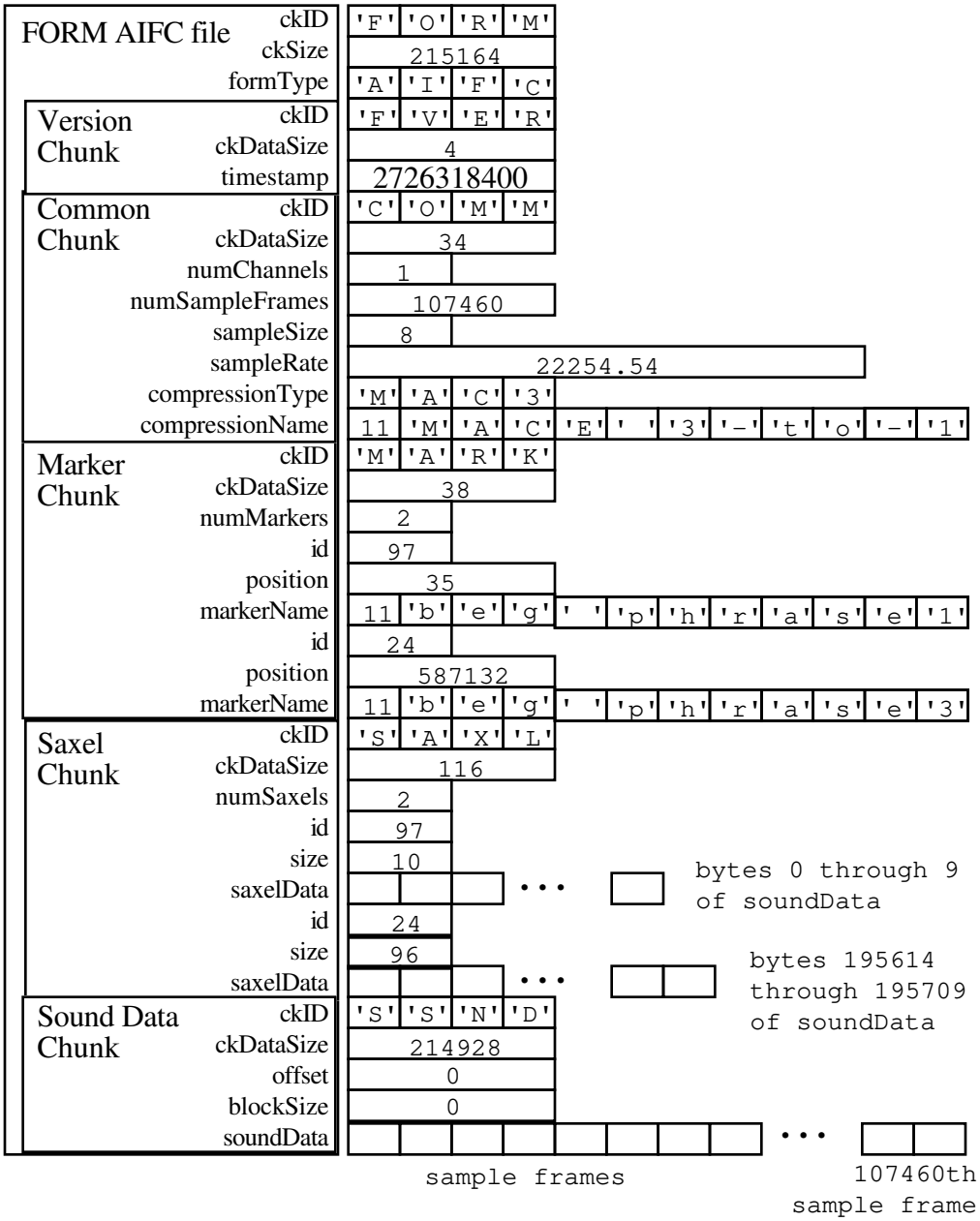
3. 16-bit stereo sound data sampled at 44.1kHz (CD quality). The sound data is not compressed.

1. A file containing approximately 4.476 seconds of 8-bit monophonic sound data sampled at 22.25454 kHz. The sound data is not compressed.

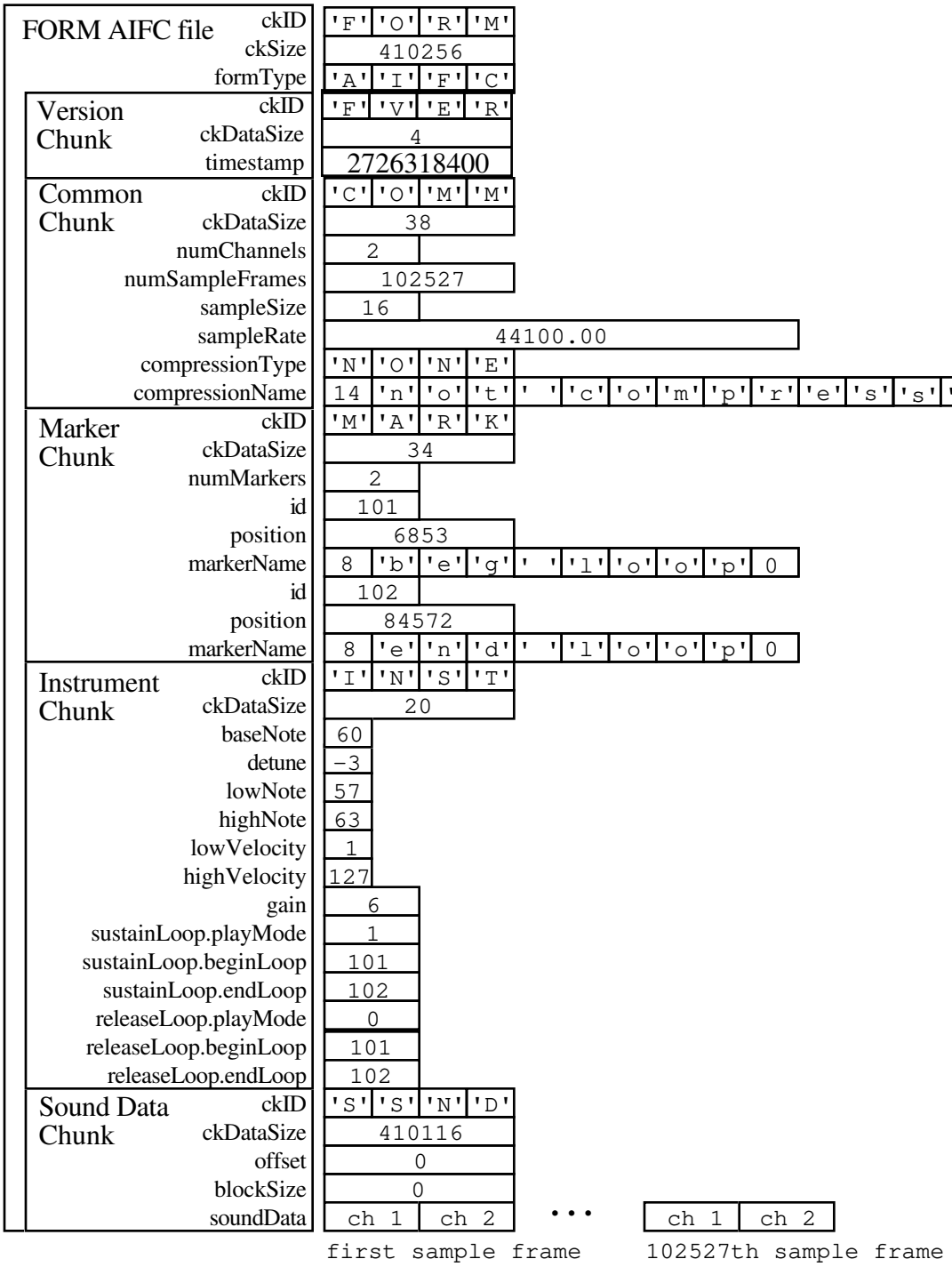


2. A file containing approximately 28.972 seconds of 8-bit sound data sampled at 22.25454 kHz and compressed by a factor of 3 using the Macintosh Audio Compression & Expansion utility.

NOTE: The Sound Accelerator Chunk (Saxel) uses the preliminary version of Saxels as defined in appendix D. This may change in the future subject to your feedback.



3. A file containing approximately 2.325 seconds of 16-bit stereo sound data sampled at 44.1kHz (CD quality). The sound data is not compressed.



Appendix B. Sending Comments to Apple

If you have suggestions for new chunks to be added to this Audio Interchange File Standard, please describe the chunk in as much detail as possible, and give an example of its use. Suggestions for new FORMs and new local chunks are welcome. When sending in suggestions, be sure to mention that your comment refers to the *Audio Interchange File Standard: "AIFF-C"* document.

Send comments to:

Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, MS: 75-3T
Cupertino, CA 95014 USA

Appendix C. Compressed Audio Encoding Format

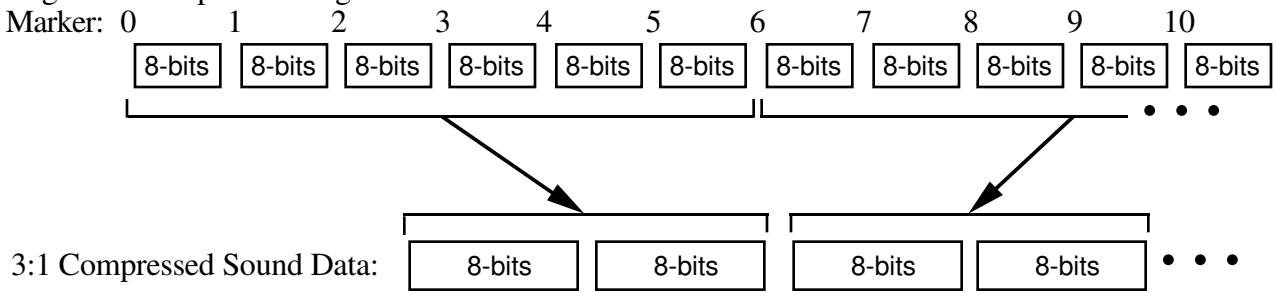
1. Encoding Formats for ACE and Macintosh compression utilities
2. Markers for ACE and Macintosh compressed sound data
3. Saxels for ACE and Macintosh compressed sound data

Encoding Formats

Encoding formats are shown for monophonic sound data. Examples of multi-channel compressed audio encoding are described below. The original sound data for the following are 8-bit linear samples.

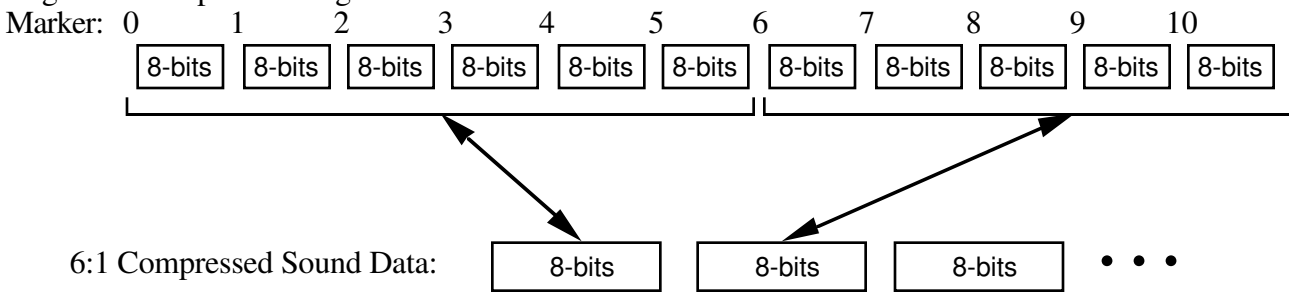
3:1 Macintosh Audio Compression & Expansion utility *Frame size = 2 bytes*

Original uncompressed single channel sound data:



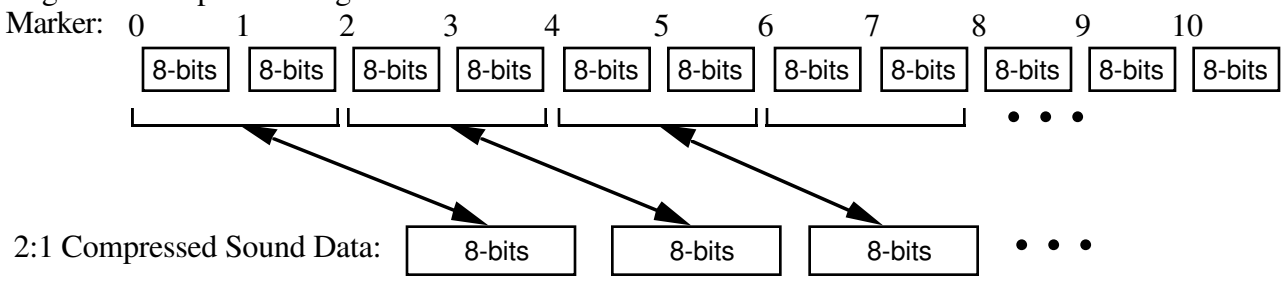
6:1 Macintosh Audio Compression & Expansion utility *Frame size = 1 byte*

Original uncompressed single channel sound data:



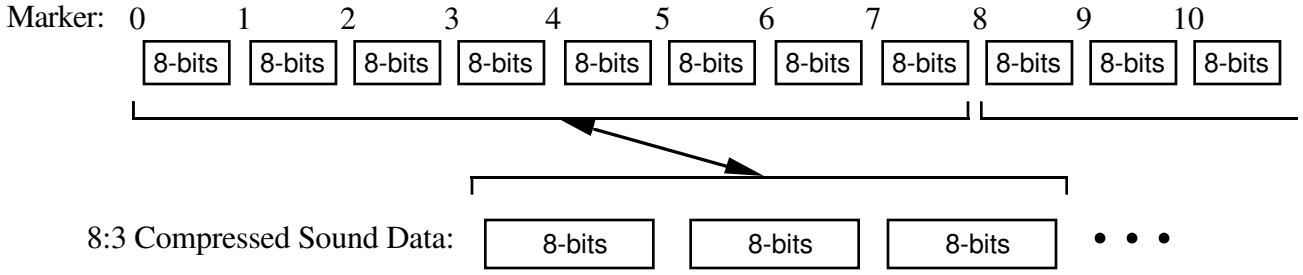
ACE 2:1 Apple IIGS utility *Frame size = 1 byte*

Original uncompressed single channel sound data:



ACE 8:3 Apple IIGS Utility *Frame size = 3 bytes*

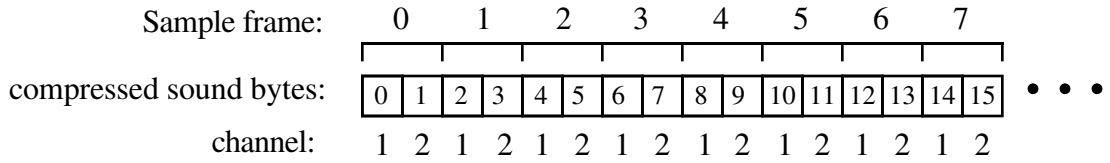
Original uncompressed single channel sound data:



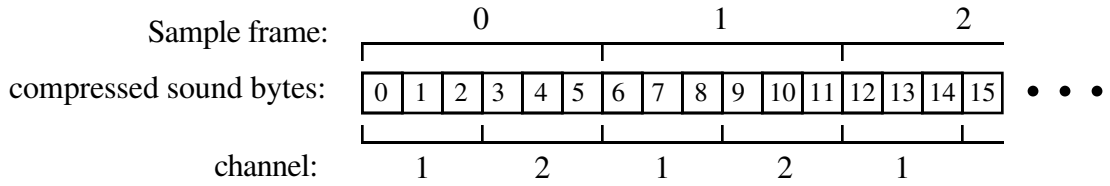
The *sample frame size* is the basic unit of a compressed data block. For the Macintosh 6:1 and the ACE 2:1 utilities - the frame size is 1 byte. For the Macintosh 3:1 utility, the frame size is 2 bytes. For the ACE 8:3 utility, the frame size is 3 bytes.

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:

Stereo: Macintosh 6:1 and IIGS ACE 2:1



Stereo: ACE 8:3



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

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) to 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.

Appendix D. Sound Accelerator (Saxel) Chunk

***** Caution *****

The use of a Sound Accelerator Chunk (Saxel) and the specific implementation of a Saxel have not yet been finalized! The draft version of this document is being prepared to get early developer feedback on the other sections of this specification. Such early feedback is considered to be more valuable than waiting for the details on this particular Chunk to be finalized. However, we do have something to say about what Saxels could look like and we would like your input on this topic as well. The following section is NOT finalized - it contains a possible, not a probable, implementation of a Saxel. Its inclusion in this specification is primarily to give you, the developer, an opportunity to learn about the purpose of a Saxel and to send us your considered feedback on this topic.

****

Saxel Definition

Audio de-compression 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 de-compressed 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. At this time, no firm decision has been made on which method to implement. The following is a tentative implementation of a variation of method (b) although there is no commitment to using this approach. We would value your feedback on this.

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.

The data format for a Saxel is inherently specific to the compression type. Here, we specify saxels for the currently supported Apple compression techniques listed below. For other compression algorithms supported by developers, other schemes for a Saxel may be employed. Applications which support compressed audio need to understand

how to process the Saxel Chunks for each compression type they support. We are primarily interested in your feedback on the following:

- **Is the Saxel Chunk as described below for Apple's audio compression algorithms suitable?**
- **What would you need in a Saxel for another compression algorithm you want to support?**

Saxel

A Saxel has the following format:

```
typedef struct {

      MarkerId        id;           /*  link accelerator data to a marker    */
      unsigned short  size;         /*  size of saxelData                    */
      char            saxelData[];  /*  algorithm-specific accelerator data  */

} Saxel;
```

id identifies 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 indicates the length in bytes of the sound accelerator data, saxelData. The data must be padded with a byte at the end as needed to make it an even number of bytes long. This pad byte, if present, is not included in *size*.

saxelData contains the specific sound accelerator data which is compression-type specific. See appendix C for a description of SaxelData formats for the Macintosh and Apple IIGS compression types.

Saxel Chunk Format

The format for the data within a Saxel Chunk is shown below.

```
#define     SaxelID   'SAXL'              /* ckID for Saxel Chunk */

typedef struct {

      ID                  ckID;           /*  'SAXL'  */
      long                ckDataSize;

      unsigned short      numSaxels;
      Saxel               saxels[];

} SaxelChunk;
```

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

numSaxels is the number of saxels in the Saxel Chunk. Multiple Saxel Chunks are allowed in a single FORM AIFC file. Since the total amount of Saxel data for a heavily-edited sound file may be quite large, it may be easier for an application to store the various Saxels independently of each other.

numSaxels, if non-zero, is followed by the saxels themselves. Since each saxel occupies an even number of bytes, the saxels are packed together with no unused bytes between them. The saxels need not be ordered in any particular manner.

The Saxel Chunk is optional. Any number of Saxel Chunks may appear in a FORM AIFC.

Saxels for ACE & Macintosh Audio Compression Types

An application which is requested to begin playback at a specific marker will first pass the data contained in *saxelData[]* to the *buffered_expansion_playback* routine without doing an audible playback. After this, the internal parameters in the expansion algorithm will have reached stability and *expansion_playback* of compressed data beginning at the marker position may begin. See the section on Markers for setting markers into compressed sound data. See appendix C for specific information on compaction methods for Macintosh and Apple IIGS compressed sounds.

The saxelData for the compression types 'ACE2', 'ACE8', 'MAC3', and 'MAC6' consist of the previous 48 *sample frames* of compressed sound data. That is, *saxelData[]* contains the 48 *sample frames* of compressed sound preceding the Marker. If the Marker position is such that there are less than 48 sample frames of compressed sound data before the expanded sample would be encountered, then the Saxel would contain the compressed sound data from the beginning up to, but not including, the compressed sample frame containing the initial sample to be played, and the Saxel *size* is set accordingly. A Saxel is not necessary for a Marker which references a sample to be expanded from the first sample frame of the compressed sound data. Here are some examples:

Macintosh 3:1 single-channel compressed sound (Frame size = 2 bytes)

Marker position	Saxel Data size	Saxel data
0 - 5	--	No Saxel for this Marker
6 - 11	2	bytes 0 - 1 of compressed sound data (1 sample frame)
12 - 17	4	bytes 0 - 3 of compressed sound data (2 sample frames)
18 - 23	6	bytes 0 - 5 of compressed sound data (3 sample frames)
. . .	. . .	. . . . .
282 - 287	94	bytes 0 - 93 of compressed sound data (47 sample frames)
288 - 293	96	bytes 0 - 95 of compressed sound data (48 sample frames)
294 - 299	96	bytes 2 - 97 of compressed sound data (48 sample frames)
300 - 305	96	bytes 4 - 99 of compressed sound data (48 sample frames)
. . .	. . .	. . . . .

Macintosh 6:1 single-channel compressed sound (Frame size = 1 byte)

Marker position	Saxel Data size	Saxel data
0 - 5	--	No Saxel for this Marker
6 - 11	1	byte 0 of compressed sound data + a zero pad byte
12 - 17	2	bytes 0 - 1 of compressed sound data
18 - 23	3	bytes 0 - 2 of compressed sound data + a zero pad byte
. . .	. . .	. . . . .
282 - 287	47	bytes 0 - 46 of compressed sound data + a zero pad byte

288 - 293	48	bytes 0 - 47 of compressed sound data
294 - 299	48	bytes 1 - 48 of compressed sound data
300 - 305	48	bytes 2 - 49 of compressed sound data
. . .	. . .	. . . . .

ACE 2:1 single-channel compressed sound (Frame size = 1 byte)

Marker position	Saxel Data size	Saxel data
0 - 1	--	No Saxel for this Marker
2 - 3	1	byte 0 of compressed sound data + a zero pad byte
4 - 5	2	bytes 0 - 1 of compressed sound data
6 - 7	3	bytes 0 - 2 of compressed sound data + a zero pad byte
. . .	. . .	. . . . .
94 - 95	47	bytes 0 - 46 of compressed sound data + a zero pad byte
96 - 97	48	bytes 0 - 47 of compressed sound data
98 - 99	48	bytes 1 - 48 of compressed sound data
100 - 101	48	bytes 2 - 49 of compressed sound data
. . .	. . .	. . . . .

ACE 8:3 single-channel compressed sound (Frame size = 3 bytes)

Marker position	Saxel Data size	Saxel data
0 - 7	--	No Saxel
8 - 15	3	bytes 0 - 2 of compressed sound data + a zero pad byte at end
16 - 23	6	bytes 0 - 5 of compressed sound data
24 - 31	9	bytes 0 - 8 of compressed sound data + a zero pad byte at end
. . .	. . .	. . . . .
376 - 383	141	bytes 0 - 140 of compressed sound data + a zero pad byte at end
384 - 391	144	bytes 0 - 143 of compressed sound data
392 - 399	144	bytes 3 - 146 of compressed sound data
400 - 407	144	bytes 6 - 149 of compressed sound data
. . .	. . .	. . . . .

Macintosh 6:1 Stereo (Frame size = 2 bytes)

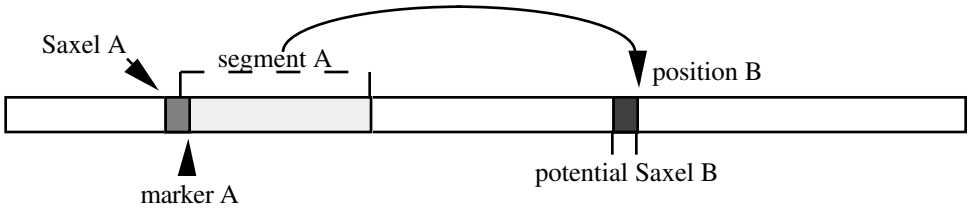
Refer to the paragraph on multichannel compressed sound storage in this appendix for clarification of which bytes are used to store the Saxel data. Remember that Marker positions reference *uncompressed* sample frames (see Marker Chunk).

Marker position	Saxel Data size	Saxel data
0 - 5	--	No Saxel
6 - 11	2	bytes 0 - 1 of compressed sound data (1 stereo sample frame)
12 - 17	4	bytes 0 - 3 of compressed sound data (2 stereo sample frames)
18 - 23	6	bytes 0 - 5 of compressed sound data (3 stereo sample frames)
. . .	. . .	. . . . .
282 - 287	94	bytes 0 - 93 of compressed sound data (47 stereo sample frames)
288 - 293	96	bytes 0 - 95 of compressed sound data (48 stereo sample frames)
294 - 299	96	bytes 2 - 97 of compressed sound data (48 stereo sample frames)
300 - 305	96	bytes 4 - 99 of compressed sound data (48 stereo sample frames)
. . .	. . .	. . . . .

Saxels and Markers

In general, whenever a Marker is created, a Saxel should be created for that Marker (except in the case where the Marker position is within the first sample frame of compressed sound data). Whenever a Marker is deleted, the Saxel for that Marker should be deleted. If a Marker exists within a portion of sound data which has been relocated, both the Marker position and Saxel for that Marker need to be updated.

If the Marker is at or near the beginning of the sound data which has been relocated, you may want to consider the following information when updating its Saxel. In the following figure, segment A of sound data containing marker A and its associated Saxel A is to be cut and pasted into the sound data stream at position B. Since Marker A is near the beginning of segment A, Saxel A contains sound data which is outside segment A. Updating Marker A to its new position would normally cause the sound data indicated by potential Saxel B to be used to refer to the new marker position.



Saxel A, however, contains the natural progression of sound which leads to marker A and creates a smooth transition to the sound data beginning at Marker A. The sound data located at potential Saxel B is not necessarily related to the sound data in segment A and may actually cause a discontinuity if used as a Saxel for the updated Marker A. Thus, to preserve sound quality for playback beginning at the updated Marker A, the original contents of Saxel A may be used for the Saxel referring to the updated Marker A.

A potential problem with this may be seen in the following two scenarios: (1) Suppose that the original Saxel A data were kept to refer to the updated Marker A which is now near position B. The user now deletes the updated Marker A, then inserts a new Marker at the same position. A new Saxel corresponding to the Marker is created using the data indicated by potential Saxel B. Sound playback beginning at the Marker position now sounds different. The reason for this may not be obvious to the user who simply cut one Marker and created another in its place. (2) A new Marker with a Saxel using approximately the sound data at potential Saxel B is created one sample position after the updated Marker A which uses the original Saxel A data to refer to it. Playback beginning at the new Marker sounds quite different than playback beginning at the updated Marker A.

The choice as to the selection of which Saxel data to use in this (hopefully unlikely) corner case depends on the contents of the sound, the position of the Marker relative to the beginning of the relocated segment and the expectations and sophistication of the user. Your application should consider these criteria and make the best choice it can, or pass the choice on to the user if required.

References

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, New York 10165

MIDI: Musical Instrument Digital Interface, Specification 1.0, the International MIDI Association.

"EA IFF 85" Standard for Interchange Format Files. Electronic Arts.
A Quick Introduction to IFF. Electronic Arts.

"8SVX" IFF 8-Bit Sampled Voice. Electronic Arts.

Inside Macintosh, Volume II. Apple Computer, Inc., Addison Wesley Publishing Company, Inc., 1986.

Apple® Numerics Manual, Addison Wesley Publishing Company, Inc., 1986.