Apple Core Audio Format Specification 1.0

Audio & Video: Audio

2006-03-08

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Tables

Introduction to Apple Core Audio Format Specification 1.0

Apple's Core Audio Format (CAF) is a file format for storing and transporting digital audio data. It simplifies the management and manipulation of many types of audio data without the file-size limitations of other audio file formats.

Who Should Read This Document?

This document is intended for anyone who needs to understand the structure of CAF files. You can use the information in this document, for example, to write a CAF parser or to extend the types of data stored in CAF files. Because CAF files offer many advantages over other audio file formats, anyone writing an application for Mac OS X that reads or writes audio files should read at least the overview chapter ("CAF File Overview" (page 9)) to gain an understanding of the features of CAF files. In addition, you need the information in this document if you want to use CAF files on other platforms.

End users of professional audio software may be interested in this document in order to learn more about the capabilities of software that supports CAF.

Organization of This Document

This document contains the following chapters:

- "CAF File Overview" (page 9) provides a brief overview of the Core Audio file format.
- "Core Audio Format Specification" (page 15) describes the CAF specification in detail.

See Also

The following documents provide additional resources:

- *Getting Started with Audio & Video* introduces the resources available for music and audio developers in Mac OS X.
- *Core Audio* describes the interfaces available to develop audio applications for Mac OS X, including instructions for writing codecs (coders/decoders), which you can use to write and read the audio data in a CAF file.

CAF File Overview

This chapter provides background information important for understanding and using Apple's Core Audio Format (CAF) files.

CAF File Advantages

Apple's Core Audio Format is a flexible, state-of-the-art file format for storing and manipulating digital audio data. It is fully supported by Core Audio APIs on Mac OS X v10.4 and later and on Mac OS X v10.3 with QuickTime 7 or later. CAF provides high performance and flexibility, and is scalable to future ultra-high resolution audio recording, editing, and playback.

CAF files have several advantages over other standard audio file formats:

- Unrestricted file size

  Whereas AIFF, AIFF-C, and WAV files are limited in size to 4 gigabytes, which might represent as little as 15 minutes of audio, CAF files use 64-bit file offsets, eliminating practical limits. A standard CAF file can hold audio data with a playback duration of hundreds of years.

- Safe and efficient recording

  Applications writing AIFF and WAV files must either update the data header's size field at the end of recording—which can result in an unusable file if recording is interrupted before the header is finalized—or they must update the size field after recording each packet of data, which is inefficient. With CAF files, in contrast, an application can append new audio data to the end of the file in a manner that allows it to determine the amount of data even if the size field in the header has not been finalized.

- Support for many data formats

  CAF files serve as wrappers for a wide variety of audio data formats. The flexibility of the CAF file structure and the many types of metadata that can be recorded enable CAF files to be used with practically any type of audio data. Furthermore, CAF files can store any number of audio channels.

- Support for many types of auxiliary data

  In addition to audio data, CAF files can store text annotations, markers, channel layouts, and many other types of information that can help in the interpretation, analysis, or editing of the audio.

- Support for data dependencies

  Certain metadata in CAF files is linked to the audio data by an edit count value. You can use this value to determine when metadata has a dependency on the audio data and, furthermore, when the audio data has changed since the metadata was written.

CAF File Structure

CAF files begin with a file header, which identifies the file type and the CAF version, followed by a series of chunks. A chunk consists of a header, which defines the type of the chunk and indicates the size of its data section, followed by the chunk data. The nature and format of the data is specific to each type of chunk.

The only two chunk types required for every CAF file are the Audio Data chunk (which, as you might have guessed, contains the audio data) and the Audio Description chunk, which specifies the audio data format.

The Audio Description chunk must be the first chunk following the file header. The Audio Data chunk can appear anywhere else in the file, unless the size of its data section has not been determined. In that case, the size field in the Audio Data chunk header is set to `-1` and the Audio Data chunk must come last in the file so that the end of the audio data chunk is the same as the end of the file. This placement allows you to determine the data section size when that information is not available in the size field.

Audio is stored in the Audio Data chunk as a sequential series of packets. An audio packet in a CAF file contains one or more frames of audio data.

CAF supports a wide range of other chunk types, which can be placed in any order in the file except first (reserved for the Audio Description chunk) or last (when the Audio Data chunk size field is set to -1). Some chunk types can be used more than once in a file. Some refer to—or are referred to by—chunks of other types.

Chunk Structure

Every chunk consists of a chunk header followed by a data section. Chunk headers contain two fields:

- A four-character code indicating the chunk's type
- A number indicating the chunk size in bytes

The format of the data in a chunk depends on the chunk type. It consists of a series of sections, typically called *fields*. The format of the audio data depends on the data type. All of the other fields in a CAF file are in big-endian (network) byte order.

Packets, Frames, and Samples

In order to understand this specification, it is important to understand the definitions of the following four terms:

- Sample

  One number for one channel of digitized audio data.

- Frame

  A set of samples representing one sample for each channel. The samples in a frame are intended to be played together (that is, simultaneously). Note that this definition might be different from the use of the term "frame" by codecs, video files, and audio or video processing applications.

- Packet

The smallest, indivisible block of data. For linear PCM (pulse-code modulated) data, each packet contains exactly one frame. For compressed audio data formats, the number of frames in a packet depends on the encoding. For example, a packet of AAC represents 1024 frames of PCM. In some formats, the number of frames per packet varies.

- Sample rate

  The number of complete frames of samples per second of noncompressed or decompressed data.

Types of Chunks

This section briefly introduces the types of chunks defined in the CAF specification. All CAF chunk types are fully described in "Core Audio Format Specification" (page 15).

Required

Every CAF file must include the following chunks:

- Audio Description chunk, which describes the audio data format for the file. This chunk must follow immediately after the CAF file header. See "Audio Description Chunk" (page 17).
- Audio Data chunk, containing the audio data for the file. If the data chunk's size isn't known, it must be the final chunk in the file. If this chunk's header specifies the size, the chunk can appear anywhere after the Audio Description chunk. See "Audio Data Chunk" (page 24).
- If the audio packets vary in size, the file must have a Packet Table chunk, which records the size of each packet. See "Packet Table Chunk" (page 25).

Channel Layout

There is one chunk that is required for all CAF files with more than two channels:

- Channel Layout chunk, which describes the role of each channel in the file. This chunk is optional for one- and two-channel files. See "Channel Layout Chunk" (page 29).

Supplementary Data

Some chunks refer to data in other, supporting chunks:

- Some compressed audio data formats require additional codec-specific data in order to decode the audio data. If the audio format requires this data, the file must have a Magic Cookie chunk. See "Magic Cookie Chunk" (page 37).
- Some chunks refer to text strings held in the Strings chunk. See "Strings Chunk" (page 37).

Markers

There are two chunks that you can use to place markers in the data file. These chunks share data types, described in "Marker Data Types" (page 39):

- Marker chunks hold individual markers. See "Marker Chunk" (page 42).
- Region chunks delineate segments of the audio data. See "Region Chunk" (page 44)

Music Metadata

There are two chunk types that store musical information:

- Instrument chunks describe aspects of the audio data needed when the audio is used by a sampler or played as an instrument. See "Instrument Chunk" (page 46).
- MIDI chunks store all of the information in a standard MIDI file. See "MIDI Chunk" (page 47).

Support For Editors

Two chunks contain data for use by audio editors:

- Overview chunks contain samples of the data useful for displaying the audio at a particular resolution. A CAF file can have any number of these; one for each resolution to be displayed. See "Overview Chunk" (page 48).
- Peak chunks list the peak amplitude in each channel and specify the frame in which that amplitude occurs. See "Peak Chunk" (page 50).

Annotations

There are two chunk types that hold annotations to the data:

- Edit Comments chunks hold time-stamped comments added when the data is edited. See "Edit Comments Chunk" (page 51).
- The Information chunk contains text strings that provide information about the audio data, such as key signature, artist, and title. See "Information Chunk" (page 52).

Identifier

One chunk type can be used to uniquely identify the data:

- The optional Unique Material Identifier (UMID) chunk provides a unique identifier for the audio data in a CAF file. There can be at most one UMID chunk in a file. See "Unique Material Identifier Chunk" (page 55).

Extending CAF

You can define your own chunk type to extend the CAF file specification. There is a chunk type defined for this purpose:

- The User-Defined chunk provides a universally unique ID (UUID) for a new chunk type. See "User-Defined Chunk" (page 56).

Extra Space

Many chunk types allow you to specify a larger chunk size than is currently needed for data in order to reserve additional space. There is also a special chunk you can use to reserve extra space in the CAF file as a whole:

- The Free chunk contains no data, but reserves space that you can use later. See "Free Chunk" (page 57).

Core Audio Format Specification

This chapter describes and specifies Apple's Core Audio Format. Refer to "CAF File Overview" (page 9) for an introduction to CAF, including information on CAF capabilities and file layout.

Important: This document uses standard C structure and enumeration declarations to specify the details of the CAF file header and CAF chunks. This is a notational convenience. The data in a CAF file is not parseable by a C compiler and does not constitute actual C structures or enumerations. For example, in a CAF file there are no pad fields to ensure correct byte alignment. Another deviation from C is that multiple "fields" in a "struct" can vary in length.

On the other hand, you can use C structures similar to those included in this document to hold the data parsed from a CAF file. The structure names (such as `CAFAudioFormat`) and the field names (such as `mChunkSize`) used in this specification are arbitrary, although many of them correspond to names used in `AudioToolbox/CAFFile.h`.

Data Types

All of the fields in a CAF file are in big-endian (network) byte order, with the exception of the audio data, which can be big- or little-endian depending on the data format. The format of the audio data is described by the Audio Description chunk.

All floating point fields in a CAF file must conform to the IEEE-754 specification. See http://grouper.ieee.org/groups/754/.

CAF File Header and Chunk Headers

The CAF file header, and the chunk header in each chunk, are required elements in every CAF file. They serve to make the file and its chunks self-describing.

CAF File Header

A CAF file begins with a simple header. The `CAFFileHeader` structure describes the file header.

```
struct CAFFileHeader
{
    UInt32  mFileType;
    UInt16  mFileVersion;
    UInt16  mFileFlags;
};
```

`mFileType`
: The file type. This value must be set to `'caff'`. You should consider only files with the `mFileType` field set to `'caff'` to be valid CAF files.

`mFileVersion`
: The file version. For CAF files conforming to this specification, the version must be set to `1`. If Apple releases a substantial revision of this specification, files compliant with that revision will have their `mFileVersion` field set to a number greater than `1`.

`mFileFlags`
: Flags reserved by Apple for future use. For CAF v1 files, must be set to `0`. You should ignore any value of this field you don't understand, and you should accept the file as a valid CAF file as long as the version and file type fields are valid.

CAF Chunk Header

Every chunk in a CAF file has a header, and each such header contains two required fields as shown in the `CAFChunkHeader` structure:

```
struct CAFChunkHeader
{
    UInt32  mChunkType;
    SInt64  mChunkSize;
};
```

`mChunkType`
: The chunk type, described as a four-character code. Apple reserves all codes that use only lowercase alphabetic characters—that is, characters in the ASCII range of `'a'–'z'` along with `' '` (space) and `'.'` (period). Application-defined chunk identifiers must include at least one character outside of this range (see "User-Defined Chunk" (page 56)).

`mChunkSize`
: The size, in bytes, of the data section for the chunk. This is the size of the chunk not including the header. Unless noted otherwise for a particular chunk type, `mChunkSize` must always be valid.

The Audio Data chunk can use the special value for `mChunkSize` of `-1` when the data section size is not known. See "Audio Data Chunk" (page 24).

CAF files can contain chunks that contain a series of entries—notably the Strings chunk, the Marker chunk, the Region chunk, and the Information chunk. The headers of these chunks can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data. The data sections of such chunks begin with a specifier for the current number of valid entries in the chunk.

CAF files can also have an optional Free chunk, used to reserve additional space for the file as a whole.

See "Free Chunk" (page 57), "Strings Chunk" (page 37), "Marker Chunk" (page 42), "Region Chunk" (page 44), and "Information Chunk" (page 52).

Required Chunks

Every CAF file must have an Audio Description chunk and an Audio Data chunk. CAF files containing variable bit rate or variable frame rate audio data must also have a Packet Table chunk.

Audio Description Chunk

The Audio Description chunk is required and must appear in a CAF file immediately following the file header. It describes the format of the audio data in the Audio Data chunk.

Audio Description Chunk Header

Table 3-1 shows the values for the fields in the Audio Description chunk header.

Table 2-1 Audio Description chunk header fields

Field	Value
`mChunkType`	`'desc'`
`mChunkSize`	`sizeof(CAFAudioFormat)`

The chunk size is fixed at `mChunkSize = sizeof(CAFAudioFormat)` to accommodate the information in the Audio Description chunk's data section.

Audio Description Chunk Data Section

The data section in the Audio Description chunk describes the format of the audio data contained within the Audio Data chunk. See "Audio Data Chunk" (page 24). For definitions needed to interpret these fields, see "Packets, Frames, and Samples" (page 10).

```
struct CAFAudioFormat
{
        Float64 mSampleRate;
        UInt32  mFormatID;
        UInt32  mFormatFlags;
        UInt32  mBytesPerPacket;
        UInt32  mFramesPerPacket;
        UInt32  mChannelsPerFrame;
        UInt32  mBitsPerChannel;
};
```

`mSampleRate`
: The number of sample frames per second of the data. You can combine this value with the frames per packet to determine the amount of time represented by a packet. This value must be nonzero.

`mFormatID`
: A four-character code indicating the general kind of data in the stream. See "mFormatID Field" (page 18). This value must be nonzero.

`mFormatFlags`
: Flags specific to each format. May be set to `0` to indicate no format flags. See "mFormatFlags Field" (page 19).

`mBytesPerPacket`
: The number of bytes in a packet of data. For formats with a variable packet size, this field is set to `0`. In that case, the file must include a Packet Table chunk "Packet Table Chunk" (page 25). Packets are

always aligned to a byte boundary. For an example of an Audio Description chunk for a format with a variable packet size, see "Compressed Audio Formats" (page 22).

`mFramesPerPacket`

The number of sample frames in each packet of data. For compressed formats, this field indicates the number of frames encoded in each packet. For formats with a variable number of frames per packet, this field is set to `0` and the file must include a Packet Table chunk "Packet Table Chunk" (page 25).

`mChannelsPerFrame`

The number of channels in each frame of data. This value must be nonzero.

`mBitsPerChannel`

The number of bits of sample data for each channel in a frame of data. This field must be set to `0` if the data format (for instance any compressed format) does not contain separate samples for each channel (see "Compressed Audio Formats" (page 22)).

The Audio Description chunk can fully describe any constant-bit-rate format that has one or more channels of the same size. For variable bit rate data, a CAF file also requires a Packet Table chunk. See "Packet Table Chunk" (page 25).

A CAF file can store any number of audio channels. The `mChannelsPerFrame` field specifies the number of channels in the data (or encoded in the data for compressed formats). For noncompressed formats, the `mBitsPerChannel` field specifies how many bits are assigned to each channel (for compressed formats, this field is `0`). The layout of the channels is described by the Channel Layout chunk ("Channel Layout Chunk" (page 29)).

mFormatID Field

The following enumeration lists some of the currently defined values for the `mFormatID` field. This list is not exhaustive.

```
enum
{
    kAudioFormatLinearPCM                = 'lpcm',
    kAudioFormatAppleIMA4                = 'ima4',
    kAudioFormatMPEG4AAC                 = 'aac ',
    kAudioFormatMACE3                    = 'MAC3',
    kAudioFormatMACE6                    = 'MAC6',
    kAudioFormatULaw                     = 'ulaw',
    kAudioFormatALaw                     = 'alaw',
    kAudioFormatMPEGLayer1               = '.mp1',
    kAudioFormatMPEGLayer2               = '.mp2',
    kAudioFormatMPEGLayer3               = '.mp3',
    kAudioFormatAppleLossless            = 'alac'
};
```

`kAudioFormatLinearPCM`

Linear PCM. Uses the PCM-related format flags discussed in "mFormatFlags Field" (page 19). See "Linear PCM" (page 20) for more information about linear PCM formats.

`kAudioFormatAppleIMA4`

Apple's implementation of IMA 4:1 ADPCM. Has no format flags. See "Compressed Audio Formats" (page 22) for more information about this and other compressed audio formats.

`kAudioFormatMPEG4AAC`

MPEG-4 AAC. The `mFormatFlags` field must contain the MPEG-4 audio object type constant indicating the specific kind of data.

`kAudioFormatMACE3`
: MACE 3:1; has no format flags.

`kAudioFormatMACE6`
: MACE 6:1; has no format flags.

`kAudioFormatULaw`
: µLaw 2:1; has no format flags.

`kAudioFormatALaw`
: aLaw 2:1; has no format flags.

`kAudioFormatMPEGLayer1`
: MPEG-1 or 2, Layer 1 audio. Has no format flags.

`kAudioFormatMPEGLayer2`
: MPEG-1 or 2, Layer 2 audio. Has no format flags.

`kAudioFormatMPEGLayer3`
: MPEG-1 or 2, Layer 3 audio (that is, MP3). Has no format flags.

`kAudioFormatAppleLossless`
: Apple Lossless; has no format flags.

mFormatFlags Field

The `mFormatFlags` field provides detailed specification for audio data formats that require it. These include linear PCM, MPEG-4 AAC, and AC-3. For audio formats that don't use formatting flags, this field must be set to `0`.

Flag bits not specified for any published format are reserved for future use. For compatibility, those flag bits should be set to `0`.

Linear PCM formatting flags can have the following values:

```
enum
{
    kCAFLinearPCMFormatFlagIsFloat          = (1L << 0),
    kCAFLinearPCMFormatFlagIsLittleEndian   = (1L << 1)
};
```

`kCAFLinearPCMFormatFlagIsFloat`
: `1` for floating point, `0` for signed integer.

`kCAFLinearPCMFormatFlagIsLittleEndian`
: `1` for little endian, `0` for big endian.

MPEG-4 AAC formatting flags use the MPEG-4 Audio Object types defined for AAC. These values are subject to revision by the MPEG-4 standards bodies.

```
enum
{
    kMP4Audio_AAC_LC_ObjectType         = 2
};
```

Linear PCM

Linear PCM (pulse-code modulated) data is the most common noncompressed audio data format. For all linear PCM formats, the `mFramesPerPacket` field equals `1` by definition. The `mBytesPerPacket` field is then equal to the number of bytes per frame. All packets are byte aligned.

The following variations of linear PCM audio should be supported by all CAF parsers:

- Any sample rate.
- Samples of 16-, 24-, and 32-bit signed integer, both big and little endian.
- Samples of 32- and 64-bit floating point, both big and little endian.

Samples of 24 bits are commonly stored within PCM CAF files in either 3 bytes per sample (packed) or 4 bytes per sample (unpacked) formats. To conform to the CAF specification, you must support both storage methods.

As an example of unpacked data, to describe 16 bit, big-endian stereo, with a sample rate of 44,100 frames per second, you would use the Audio Description field values in Table 3-2.

Table 2-2 Audio Description field values for big-endian unpacked 16-bit PCM

Field	Value
`mSampleRate`	44100.
`mFormatID`	`kAudioFormatLinearPCM`
`mFormatFlags`	`0` (big-endian integer)
`mChannelsPerFrame`	2
`mBitsPerChannel`	16
`mFramesPerPacket`	1
`mBytesPerPacket`	4

In the packed case, each 24 bit sample takes up 3 bytes in the file. For example, to describe 24 bit, little-endian stereo, with a sample rate of 48,000 frames per second, you would use the Audio Description field values in Table 3-3.

Table 2-3 Audio Description field values for little-endian packed 24-bit PCM

Field	Value
`mSampleRate`	48000.
`mFormatID`	`kAudioFormatLinearPCM`
`mFormatFlags`	`kCAFLinearPCMFormatFlagIsLittleEndian`
`mChannelsPerFrame`	2
`mBitsPerChannel`	24

Field	Value
mFramesPerPacket	1
mBytesPerPacket	6

In the unpacked case, the 24 bits are aligned high within the 4 byte field so that a parser can treat the value as if it were 32 bit integer with the lowest (or least significant) 8 bits all zero). On disk, the little-endian version of this data format looks like this:

```
OO LL XX MM
```

where `MM` is the most significant byte and `LL` is the least significant.

A big-endian version of 24-bit PCM audio in 4 bytes looks like this:

```
MM XX LL OO
```

The Audio Description chunk for this format is the same as for the packed version (Table 3-3), except that the `mBytesPerPacket` field is set to 8 rather than 6.

To describe floating point samples, you have to add the `kCAFLinearPCMFormatFlagIsFloat` flag to the `mFormatFlags` field. For example, to describe 4 channels of little-endian 64-bit floating point samples with a sample rate of 96,000 frames per second, you would use the Audio Description chunk field values in Table 3-4.

Table 2-4 Audio Description field values for floating-point 64-bit PCM

Field	Value
mSampleRate	96000.
mFormatID	kAudioFormatLinearPCM
mFormatFlags	kCAFLinearPCMFormatFlagIsFloat | kCAFLinearPCMFormatFlagIsLittleEndian
mChannelsPerFrame	4
mBitsPerChannel	64
mFramesPerPacket	1
mBytesPerPacket	32

You can also use CAF files to store non-byte-aligned PCM formats, such as 12-bit or 18-bit PCM. To do so, you should

1. Pack the data within a byte-aligned sample width.
2. High-align the samples within the enclosing byte-aligned width.

For example, 12-bit PCM data should be packed (high-aligned) within a 2-byte (16-bit) word, allowing the CAF parser to parse the sample data using the same algorithms as used for 16-bit data.

In this case the Audio Description chunk for the 12-bit data would be identical to a chunk for 16-bit data, except that the `mBitsPerChannel` field would be set to 12 rather than 16.

Pulse Width Modulation

In the Pulse Width Modulation (PWM) format (also known as 1-bit audio), each sample is one bit. This is the data format used for Super Audio CD (SA-CD; see http://www.superaudio-cd.com/). Although CAF does not define a format ID constant for a PWM format, it is instructive to look at how PWM data would be stored.

The sample rate for a Super Audio CD bit stream is 2,822,400 frames per second. In a CAF file with PWM data there would be no format flags, 1 bit per channel, and 8 frames per packet. Therefore, for two channels (stereo), there would be 2 bytes per packet (1 byte for each channel in the file).

Stereo PWM is packed as follows (in binary):

```
LLLLLLLL RRRRRRRR
```

where `L` is a bit for the left channel and `R` is a bit for the right channel. Therefore, the first `L` bit together with the first `R` bit constitute the first frame.

Similarly, for 6 channels there would be 6 bytes per packet and 8 frames per packet, packed as follows:

```
11111111 22222222 33333333 44444444 55555555 66666666
```

As is true for the data in all CAF files, the PWM data is byte aligned.

Compressed Audio Formats

In compressed audio formats, the packets are opaque and cannot be parsed without first being decompressed by a codec. For such formats, the `mSampleRate` field indicates the number of sample frames per second of the decompressed data and the `mFramesPerPacket` field indicates the number of frames encoded in each compressed packet. In addition, for compressed formats the `mBitsPerChannel` field is always `0`. All packets in CAF files must be byte aligned.

For example, the IMA4 data format encodes 64-sample frames into a single packet with a constant bit rate of 34 bytes per channel. To describe a CAF file of 2 channel IMA4 data with a sampling rate of 44,100 frames per second, you would use the Audio Description field values in Table 3-5.

Table 2-5 Audio Description field values for 2 channel IMA4

Field	Value
`mSampleRate`	44100.
`mFormatID`	`kAudioFormatAppleIMA4`
`mFormatFlags`	0
`mChannelsPerFrame`	2
`mBitsPerChannel`	0

Field	Value
mFramesPerPacket	64
mBytesPerPacket	68 (= mChannelsPerFrame * 34)

In this example, the mBitsPerChannel field is 0, indicating that this is a compressed format. The mBytesPerPacket field reflects the constant number of bytes per channel (34) and the number of frames per packet (64 in this case).

For a compressed audio format with a variable bit rate, the mBytesPerPacket field is 0, indicating that the number of bytes per packet is variable. In this case, a Packet Table chunk ("Packet Table Chunk" (page 25)) is required.

For example, the MPEG-4 Advanced Audio Coding (AAC) data format uses a variable bit rate but a constant number of frames per packet. To describe a CAF file of 2 channel Low Complexity Audio Object format AAC data with a sampling rate of 44,100 frames per second (for the decompressed data), you would use the Audio Description field values in Table 3-6.

Table 2-6 Audio Description field values for 2 channel AAC

Field	Value
mSampleRate	44100.
mFormatID	kAudioFormatMPEG4AAC
mFormatFlags	kMP4Audio_AAC_LC_ObjectType
mChannelsPerFrame	2
mBitsPerChannel	0
mFramesPerPacket	1024
mBytesPerPacket	0

In this example, the mBitsPerChannel field is 0, indicating that this is a compressed format, and the mBytesPerPacket field is 0, indicating a variable bit rate.

Note that, as long as the format has a constant number of frames per packet, you can calculate the duration of each packet by dividing the mSampleRate value by the mFramesPerPacket value.

Some compressed formats vary the number of frames per packet. In this case, you must set the mFramesPerPacket field to 0 (in addition to the mBitsPerChannel field, which is 0 for all compressed formats).

Audio Data Chunk

Every CAF file must have exactly one Audio Data chunk. Whereas other chunks contain data that help to characterize or interpret the audio, this is the chunk in a CAF file that contains the actual audio data. If its size is specified, this chunk can be placed anywhere following the Audio Description chunk. If its size is not specified, the Audio Data chunk must be last in the file.

Audio Data Chunk Header

Table 3-7 shows the values for the fields in the Audio Data chunk header.

Table 2-7 Audio Data chunk header fields

Field	Value
`mChunkType`	`'data'`
`mChunkSize`	Size of data section in bytes, or `-1` if unknown.

An `mChunkSize` value of `-1` indicates that the size of the data section for this chunk is unknown. In this case, the Audio Data chunk must appear last in the file so that the end of the Audio Data chunk is the same as the end of the file. This placement allows you to determine the data section size.

It is highly recommended that, after recording or modifying the audio data, you finalize the CAF file by updating the `mChunkSize` field to reflect the size of the Audio Data chunk's data section. When you read a CAF file whose audio data section size is not specified, you should determine the size and update the `mChunkSize` value for the Audio Data chunk.

If the Audio Data chunk is not the last chunk in a CAF file, the `mChunkSize` field must contain the size of the chunk's data section for the file to be valid.

Immediately following the Audio Data chunk's header is the audio data section.

Audio Data Chunk Data Section

The data section in an Audio Data chunk contains audio data in the format specified by the Audio Description chunk. See "Audio Description Chunk" (page 17).

The Audio Data chunk's data section has an edit count field followed by the audio data for the file. The `CAFData` structure describes the data section for this chunk.

```
struct CAFData
{
    UInt32  mEditCount;                     //initially set to 0
    UInt8   mData [kVariableLengthArray];
};
```

`mEditCount`
: The modification status of the data section. You should initially set this field to `0`, and should increment it each time the audio data in the file is modified.

`mData`
: The audio data for the CAF file, in the format specified by the Audio Description chunk.

You can compare the value of `mEditCount` to the corresponding value in a dependent chunk, such as the "Overview Chunk" (page 48) or "Peak Chunk" (page 50).

This document does not address the specifics of the data formats specified by the Audio Description chunk. Refer to specifications issued by the appropriate standards body or industry entity for information on a specific audio data format.

Packet Table Chunk

CAF files that contain variable bit-rate (VBR) or variable frame-rate (VFR) audio data contain audio packets of varying size. Such files must have exactly one Packet Table chunk to specify the size of each packet.

You can identify CAF files containing VBR or VFR audio by their Audio Description chunk. In such files, one or both of the `mBytesPerPacket` and `mFramesPerPacket` fields in the Audio Description chunk has a value of `0`. See "Audio Description Chunk" (page 17).

The content of the Packet Table chunk describes, and therefore depends on, the content of the Audio Data chunk. See "Audio Data Chunk" (page 24). The packet table must always reflect current state of the audio data in a CAF file.

A CAF file with constant packet size can still include a Packet Table chunk in order to record certain information about frames (see "Packet Table Description" (page 25)).

Packet Table Chunk Header

Table 3-8 shows the values for the fields in the Packet Table chunk header.

Table 2-8 Packet Table chunk header fields

Field	Value
`mChunkType`	`'pakt'`
`mChunkSize`	Must always be valid

For a CAF file with variable packet sizes, the value for `mChunkSize` can be greater than the actual valid content of the packet table chunk. The Packet Table description indicates the number of valid entries in the Packet Table (see "Packet Table Description" (page 25)). In the case of a CAF file with constant packet size, the value for `mChunkSize` should be 24 bytes—just enough to contain the Packet Table description itself.

Packet Table Description

This chunk has a descriptive section for the packet table itself. It appears immediately after the chunk header. The `CAFPacketTableHeader` structure describes it:

```
struct CAFPacketTableHeader
{
    SInt64  mNumberPackets;
    SInt64  mNumberValidFrames;
    SInt32  mPrimingFrames;
    SInt32  mRemainderFrames;
```

```
};
```

`mNumberPackets`
: The total number of packets of audio data described in the packet table. This value must always be valid.

  For a CAF file with variable packet sizes, this value should reflect the actual number of packets in the Audio Data chunk. In a CAF file with constant packet size, and therefore no packet table, this field should be set to `0`.

`mNumberValidFrames`
: The total number of audio frames encoded in the file. The duration of the audio in the file is this value divided by the sample rate specified in the file's Audio Description chunk. See "Audio Description Chunk" (page 17). The value of this field must always be valid.

`mPrimingFrames`
: The number of frames for priming or processing latency for a compressed audio format. For example, MPEG-AAC codecs typically have a latency of 2112 frames. The number of priming frames can be useful for any CAF file containing compressed audio, whether or not the packets vary in size.

`mRemainderFrames`
: The number of unused frames in the CAF file's final packet; that is, the number of frames that should be trimmed from the output of the last packet when decoding.

  For example, an AAC file may have only 313 frames containing audio data in its final packet. AAC files hold 1024 frames per packet. The value for `mRemainderFrames` is then 1024 – 313 = 711.

The `mNumberPackets` value is specified only when the chunk contains a packet table—that is, when the CAF file contains variable-sized packets. On the other hand, regardless of whether its packets vary in size or not, any CAF file can use the `mNumberValildFrames`, `mPrimingFrames`, and `mRemainderFrames` fields.

Packet Table Chunk Data Section

The Packet Table chunk's data section lists information about variable-sized packets in the file's Audio Data chunk. See "Audio Data Chunk" (page 24).

For a given CAF file, depending on the file's audio format, packets can vary in size because of a variable bit rate (variable bytes per packet), a variable number of frames per packet, or both.

The following list of these three audio format types includes the corresponding values for `mBytesPerPacket` and `mFramesPerPacket` present in the Audio Description chunk. See "Audio Description Chunk" (page 17):

- Variable bit rate, constant number of frames per packet (such as AAC and variable-bit-rate MP3): `mBytesPerPacket` is zero, `mFramesPerPacket` is nonzero.

  The Packet Table chunk data section contains single-number entries that describe the size, in bytes, of each packet in the Audio Data chunk.

- Variable number of frames per packet, constant bit rate: `mBytesPerPacket` is nonzero; `mFramesPerPacket` is zero.

  The Packet Table chunk data section contains single-number entries that describe the number of frames represented by each packet in the Audio Data chunk.

- Variable bit rate, variable number of frames per packet (such as Ogg Vorbis): `mBytesPerPacket` is zero, `mFramesPerPacket` is zero.

  The Packet Table chunk data section contains ordered-pair entries. The first number in each pair is the packet size, in bytes; the second is the number of frames per packet.

The numbers describing the size of packets or frames per packet are encoded as variable-length integers. In this encoding scheme, each byte contains 7 bits of the binary integer and a 1-bit continuation flag—the high-order bit in each byte is used to indicate whether the number is continued in the next byte. The lowest-order byte in any given integer is therefore the first one for which the high-order bit is not set; that is, the first byte that has a value less than 128 holds the last 7 bits in the integer. Table 3-9 gives some examples of encoded integers.

Table 2-9 Sample variable-length encoded integers

Packet size	Integer encoding (hexadecimal)	Integer encoding (binary)
1	0x01	0000 0001
17	0x11	0001 0001
127	0x7F	0111 1111
128	0x81 0x00	1000 0001 0000 0000
130	0x81 0x02	1000 0001 0000 0010
257	0x82 0x01	1000 0010 0000 0001
16383	0xFF 0x7F	1111 1111 0111 1111
16384	0x81 0x80 0x00	1000 0001 1000 0000 0000 0000

Thus, the data section contains a simple list of numbers or a list of ordered pairs of numbers. In all cases, variable-length integers are used to describe each packet.

Constant Bit Rate Format

A Packet Table chunk may be used with a constant bit rate (constant frames per packet and constant bytes per packet) format to provide information about either of the following:

- Any latency due to the nature of the codec (see the discussion of the `mPrimingFrames` field in "Packet Table Description" (page 25)).
- Any remainder frames. Remainder frames occur when the total number of frames in the audio data is not evenly divisible by the frames per packet specified for the file. See the discussion of the `mFramesPerPacket` field in "Audio Description Chunk Data Section" (page 17) and the discussion of the `mRemainderFrames` field in "Packet Table Description" (page 25).

For either of these cases, no packet table data is needed, so set the `mNumberPackets` field to `0`. The size of the packet table is therefore the size of the packet table header structure.

As an example of the second use, the IMA format encodes samples into packets containing 64 sample frames each. If the audio data is not equally divisible by 64 frames, then the last packet of IMA content decodes to less samples than the 64 that are presented by the packet. In this case, the Packet Table header is used to indicate the total number of frames in the file and the number of remainder frames. For example, if there are 5 remainder frames, you would set the fields of the Packet Table header as shown in Table 3-10.

Table 2-10 Packet Table header for an IMA file with 5 remainder frames

Field	Value
mNumberPackets	0
mNumberValidFrames	`numFramesInFile (= (mFramesPerPacket * (sizeOfAudioData) / mBytesPerPacket)-5)`
mPrimingFrames	0
mRemainderFrames	`59 (= mFramesPerPacket - 5)`

Variable Bit Rate, Constant Frames per Packet

The Packet Table chunk is required for compressed data formats with a variable bit rate (`mBytesPerPacket` is set to `0`) and a constant number of frames per packet (`mFramesPerPacket` is nonzero). (See "Audio Description Chunk Data Section" (page 17) for more information about these header fields.)

In this case, the packet table data contains one variable-length integer for each packet specifying the packet's size in bytes. See "Packet Table Chunk Data Section" (page 26) for an explanation of variable-length integers.

For example, because AAC has a latency of 2112 frames, an AAC encoding of 3074 sample frames requires a total of 6 packets (AAC has 1024 frames per packet). The fields of the Packet Table header for this example are as shown in Table 3-11.

Table 2-11 Packet Table header for an AAC file

Field	Value
`mNumberPackets`	6
`mNumberValidFrames`	3074
`mPrimingFrames`	2112
`mRemainderFrames`	958 (= 1024 (`mFramesPerPacket`) – 66)

As shown in Table 3-12, the first two packets contain only priming frames; these frames do not output any valid audio data. The third packet contains the final 64 priming frames and then outputs 960 frames of audio data. The following two packets contain 1024 sample frames of valid audio data apiece. (There would normally be many more 1024-frame packets than the two in this example.) The last packet contains the final 66 sample frames of audio data followed by 958 remainder frames (which should be trimmed from the output).

Table 2-12 Frames of valid audio data per packet for an AAC file of 3074 frames

Packet	1	2	3	4	5	6
Valid Frames	0	0	960	1024	1024	66
Total Frames	1024	1024	1024	1024	1024	1024

Note that the Audio Description chunk would specify this file as having a constant 1024 frames per packet. The priming and trailing frame counts can be used to determine how to trim the audio output of the file when the data is decoded.

Following this packet table header is the packet table itself, which in this example would consist of 6 variable sized integers that describe the number of bytes for each of the 6 packets.

Channel Layout

The channel layout chunk is required for all CAF files that have more than two channels (unless there is no meaning or ordering of the channels in the file). There is no default assumed ordering of channels in a file with more than two channels. The channel layout chunk is optional for a CAF file with one or two channels. For a CAF file with one or two channels and no channel layout chunk, you can assume that a one-channel file represents monaural data and a two-channel file represents stereo with the left-channel sample first in each frame.

Channel Layout Chunk

The Channel Layout chunk describes the order and role of each channel in a CAF file. It is especially useful for any CAF file with more than two audio channels but can also provide important information for one- and two-channel files. For example, when a user converts a stereo or multichannel audio file to a set of one-channel files, the Channel Layout chunk can indicate the role of each one-channel file.

In the Audio Data chunk ("Audio Data Chunk" (page 24)) of an uncompressed audio CAF file, a sample for each channel appears in sequence in each frame. The number of channels per frame and the number of bits per channel are specified in the Audio Description chunk (see "Audio Description Chunk Data Section" (page 17)). The Channel Description chunk specifies the order in which the channel data appears in the audio data chunk.

Channel Layout Chunk Header

Table 3-13 shows the values for the fields in the Channel Layout chunk header.

Table 2-13 Channel Layout chunk header fields

Field	Value
`mChunkType`	`'chan'`
`mChunkSize`	Must always be valid

The `mChunkSize` field must be set to the size of the chunk's data section and must always be valid.

Channel Layout Chunk Data Section

The Channel Layout chunk data section begins with a tag that indicates the nature of the data in the chunk, followed by the data, as shown in the `CAFChannelLayout` structure.

```
struct CAFChannelLayout
{
    UInt32                  mChannelLayoutTag;
    UInt32                  mChannelBitmap;
    UInt32                  mNumberChannelDescriptions;
    CAFChannelDescription   mChannelDescriptions[kVariableLengthArray];
};
```

`mChannelLayoutTag`
: A tag that indicates the type of layout used, as described in "Channel Layout Tags" (page 30).

`mChannelBitmap`
: A bitmap that describes which channels are present. The order of the channels is the same as the order of the bits; that is, the lowest-order bit that is set corresponds to the first channel of the file, and so on. The number of set bits is the number of channels, which must equal the number of channels in the file. This bit-field technique is used both in WAV files and in the USB Audio Specification. See "Channel Bitmaps" (page 30) for bit assignments.

`mNumberChannelDescriptions`
: The number of channel descriptions in the `mChannelDescriptions` array. If this number is `0`, then this is the last field in the structure.

`mChannelDescriptions`
: An array of `CAFChannelDescription` structures ("Channel Description" (page 34)) that describe the layout of the channels. This field is not present if the `mNumberChannelDescriptions` field is `0`.

Channel Bitmaps

The significance of the bits in the `mChannelBitmap` field is specified in the following enumeration:

```
enum
{
    kCAFChannelBit_Left                 = (1<<0),
    kCAFChannelBit_Right                = (1<<1),
    kCAFChannelBit_Center               = (1<<2),
    kCAFChannelBit_LFEScreen            = (1<<3),
    kCAFChannelBit_LeftSurround         = (1<<4),   // WAVE: "Back Left"
    kCAFChannelBit_RightSurround        = (1<<5),   // WAVE: "Back Right"
    kCAFChannelBit_LeftCenter           = (1<<6),
    kCAFChannelBit_RightCenter          = (1<<7),
    kCAFChannelBit_CenterSurround       = (1<<8),   // WAVE: "Back Center"
    kCAFChannelBit_LeftSurroundDirect   = (1<<9),   // WAVE: "Side Left"
    kCAFChannelBit_RightSurroundDirect  = (1<<10), // WAVE: "Side Right"
    kCAFChannelBit_TopCenterSurround    = (1<<11),
    kCAFChannelBit_VerticalHeightLeft   = (1<<12), // WAVE: "Top Front Left"
    kCAFChannelBit_VerticalHeightCenter = (1<<13), // WAVE: "Top Front Center"
    kCAFChannelBit_VerticalHeightRight  = (1<<14), // WAVE: "Top Front Right"
    kCAFChannelBit_TopBackLeft          = (1<<15),
    kCAFChannelBit_TopBackCenter        = (1<<16),
    kCAFChannelBit_TopBackRight         = (1<<17)
};
```

Channel Layout Tags

Channel layouts can be described by a code in the `mChannelLayoutTag` field.

A value of `kCAFChannelLayoutTag_UseChannelDescriptions` (`== 0`) indicates there is no standard description for the ordering or use of channels in the file, so that channel descriptions are used instead. In this case, the number of channel descriptions (`mNumberChannelDescriptions`) must equal the number of channels contained in the file. The channel descriptions follow the `mNumberChannelDescriptions` field; see "Channel Description" (page 34).

A value of `kCAFChannelLayoutTag_UseChannelBitmap` (== 0x10000) indicates that the Channel Layout chunk uses a bitmap (in the `mChannelBitmap` field) to describe which channels are present.

A value greater than `0x10000` indicates one of the layout tags listed below in this section. Each channel layout tag has two parts:

- The low 16 bits represents the number of channels described by the tag.
- The high 16 bits indicates a specific ordering of the channels.

For example, the tag `kCAFChannelLayoutTag_Stereo` is defined as `((101<<16) | 2 )` and indicates a two-channel stereo, ordered left as the first channel, right as the second.

Current values for this code are listed in the following enumeration:

```
enum {
    kCAFChannelLayoutTag_UseChannelDescriptions = (0<<16) | 0,
        // use the array of AudioChannelDescriptions to define the mapping.

    kCAFChannelLayoutTag_UseChannelBitmap       = (1<<16) | 0,
        // use the bitmap to define the mapping.

// 1 Channel Layout
    kCAFChannelLayoutTag_Mono                   = (100<<16) | 1,
        // a standard mono stream

// 2 Channel layouts
    kCAFChannelLayoutTag_Stereo                 = (101<<16) | 2,
        // a standard stereo stream (L R)

    kCAFChannelLayoutTag_StereoHeadphones       = (102<<16) | 2,
        // a standard stereo stream (L R) - implied headphone playback

    kCAFChannelLayoutTag_MatrixStereo           = (103<<16) | 2,
        // a matrix encoded stereo stream (Lt, Rt)

    kCAFChannelLayoutTag_MidSide                = (104<<16) | 2,
        // mid/side recording

    kCAFChannelLayoutTag_XY                     = (105<<16) | 2,
        // coincident mic pair (often 2 figure 8's)

    kCAFChannelLayoutTag_Binaural               = (106<<16) | 2,
        // binaural stereo (left, right)

// Symmetric arrangements - same distance between speaker locations
    kCAFChannelLayoutTag_Ambisonic_B_Format     = (107<<16) | 4,
        // W, X, Y, Z

    kCAFChannelLayoutTag_Quadraphonic           = (108<<16) | 4,
        // front left, front right, back left, back right
```

```
    kCAFChannelLayoutTag_Pentagonal             = (109<<16) | 5,
        // left, right, rear left, rear right, center

    kCAFChannelLayoutTag_Hexagonal              = (110<<16) | 6,
        // left, right, rear left, rear right, center, rear

    kCAFChannelLayoutTag_Octagonal              = (111<<16) | 8,
        // front left, front right, rear left, rear right,
        // front center, rear center, side left, side right

    kCAFChannelLayoutTag_Cube                   = (112<<16) | 8,
        // left, right, rear left, rear right
        // top left, top right, top rear left, top rear right

//  MPEG defined layouts
    kCAFChannelLayoutTag_MPEG_1_0   = kCAFChannelLayoutTag_Mono,    //  C
    kCAFChannelLayoutTag_MPEG_2_0   = kCAFChannelLayoutTag_Stereo,  //  L R
    kCAFChannelLayoutTag_MPEG_3_0_A = (113<<16) | 3,         // L R C
    kCAFChannelLayoutTag_MPEG_3_0_B = (114<<16) | 3,         // C L R
    kCAFChannelLayoutTag_MPEG_4_0_A = (115<<16) | 4,         // L R C Cs
    kCAFChannelLayoutTag_MPEG_4_0_B = (116<<16) | 4,         // C L R Cs
    kCAFChannelLayoutTag_MPEG_5_0_A = (117<<16) | 5,         // L R C Ls Rs
    kCAFChannelLayoutTag_MPEG_5_0_B = (118<<16) | 5,         // L R Ls Rs C
    kCAFChannelLayoutTag_MPEG_5_0_C = (119<<16) | 5,         // L C R Ls Rs
    kCAFChannelLayoutTag_MPEG_5_0_D = (120<<16) | 5,         // C L R Ls Rs
    kCAFChannelLayoutTag_MPEG_5_1_A = (121<<16) | 6,         // L R C LFE Ls Rs
    kCAFChannelLayoutTag_MPEG_5_1_B = (122<<16) | 6,         // L R Ls Rs C LFE
    kCAFChannelLayoutTag_MPEG_5_1_C = (123<<16) | 6,         // L C R Ls Rs LFE
    kCAFChannelLayoutTag_MPEG_5_1_D = (124<<16) | 6,         // C L R Ls Rs LFE
    kCAFChannelLayoutTag_MPEG_6_1_A = (125<<16) | 7,         // L R C LFE Ls Rs Cs
    kCAFChannelLayoutTag_MPEG_7_1_A = (126<<16) | 8,         // L R C LFE Ls Rs Lc Rc
    kCAFChannelLayoutTag_MPEG_7_1_B = (127<<16) | 8,         // C Lc Rc L R Ls Rs LFE
    kCAFChannelLayoutTag_MPEG_7_1_C = (128<<16) | 8,         // L R C LFE Ls R Rls Rrs

    kCAFChannelLayoutTag_Emagic_Default_7_1 = (129<<16) | 8,
        //  L R Ls Rs C LFE Lc Rc


    kCAFChannelLayoutTag_SMPTE_DTV          = (130<<16) | 8,
        //  L R C LFE Ls Rs Lt Rt
        //  (kCAFChannelLayoutTag_ITU_5_1 plus a matrix encoded stereo mix)

//  ITU defined layouts
    kCAFChannelLayoutTag_ITU_1_0    = kCAFChannelLayoutTag_Mono,    //  C
    kCAFChannelLayoutTag_ITU_2_0    = kCAFChannelLayoutTag_Stereo,  //  L R

    kCAFChannelLayoutTag_ITU_2_1    = (131<<16) | 3,         // L R Cs
    kCAFChannelLayoutTag_ITU_2_2    = (132<<16) | 4,         // L R Ls Rs
    kCAFChannelLayoutTag_ITU_3_0    = kCAFChannelLayoutTag_MPEG_3_0_A,  //  L R C
    kCAFChannelLayoutTag_ITU_3_1    = kCAFChannelLayoutTag_MPEG_4_0_A,  //  L R C Cs

    kCAFChannelLayoutTag_ITU_3_2    = kCAFChannelLayoutTag_MPEG_5_0_A,  //  L R C Ls Rs
    kCAFChannelLayoutTag_ITU_3_2_1  = kCAFChannelLayoutTag_MPEG_5_1_A,
                                                        //  L R C LFE Ls Rs
    kCAFChannelLayoutTag_ITU_3_4_1  = kCAFChannelLayoutTag_MPEG_7_1_C,
                                                    //  L R C LFE Ls Rs Rls Rrs
```

```
// DVD defined layouts
    kCAFChannelLayoutTag_DVD_0  = kCAFChannelLayoutTag_Mono,    // C (mono)
    kCAFChannelLayoutTag_DVD_1  = kCAFChannelLayoutTag_Stereo,  // L R
    kCAFChannelLayoutTag_DVD_2  = kCAFChannelLayoutTag_ITU_2_1, // L R Cs
    kCAFChannelLayoutTag_DVD_3  = kCAFChannelLayoutTag_ITU_2_2, // L R Ls Rs
    kCAFChannelLayoutTag_DVD_4  = (133<<16) | 3,                // L R LFE
    kCAFChannelLayoutTag_DVD_5  = (134<<16) | 4,                // L R LFE Cs
    kCAFChannelLayoutTag_DVD_6  = (135<<16) | 5,                // L R LFE Ls Rs
    kCAFChannelLayoutTag_DVD_7  = kCAFChannelLayoutTag_MPEG_3_0_A,// L R C
    kCAFChannelLayoutTag_DVD_8  = kCAFChannelLayoutTag_MPEG_4_0_A,// L R C Cs
    kCAFChannelLayoutTag_DVD_9  = kCAFChannelLayoutTag_MPEG_5_0_A,// L R C Ls Rs
    kCAFChannelLayoutTag_DVD_10 = (136<<16) | 4,                // L R C LFE
    kCAFChannelLayoutTag_DVD_11 = (137<<16) | 5,                // L R C LFE Cs
    kCAFChannelLayoutTag_DVD_12 = kCAFChannelLayoutTag_MPEG_5_1_A,// L R C LFE Ls Rs
    // 13 through 17 are duplicates of 8 through 12.
    kCAFChannelLayoutTag_DVD_13 = kCAFChannelLayoutTag_DVD_8,   // L R C Cs
    kCAFChannelLayoutTag_DVD_14 = kCAFChannelLayoutTag_DVD_9,   // L R C Ls Rs
    kCAFChannelLayoutTag_DVD_15 = kCAFChannelLayoutTag_DVD_10,  // L R C LFE
    kCAFChannelLayoutTag_DVD_16 = kCAFChannelLayoutTag_DVD_11,  // L R C LFE Cs
    kCAFChannelLayoutTag_DVD_17 = kCAFChannelLayoutTag_DVD_12,  // L R C LFE Ls Rs
    kCAFChannelLayoutTag_DVD_18 = (138<<16) | 5,                // L R Ls Rs LFE
    kCAFChannelLayoutTag_DVD_19 = kCAFChannelLayoutTag_MPEG_5_0_B,// L R Ls Rs C
    kCAFChannelLayoutTag_DVD_20 = kCAFChannelLayoutTag_MPEG_5_1_B,// L R Ls Rs C LFE

// These layouts are recommended for Mac OS X's AudioUnit use
        // These are the symmetrical layouts
    kCAFChannelLayoutTag_AudioUnit_4= kCAFChannelLayoutTag_Quadraphonic,
    kCAFChannelLayoutTag_AudioUnit_5= kCAFChannelLayoutTag_Pentagonal,
    kCAFChannelLayoutTag_AudioUnit_6= kCAFChannelLayoutTag_Hexagonal,
    kCAFChannelLayoutTag_AudioUnit_8= kCAFChannelLayoutTag_Octagonal,
        // These are the surround-based layouts
    kCAFChannelLayoutTag_AudioUnit_5_0  = kCAFChannelLayoutTag_MPEG_5_0_B,
                                                            // L R Ls Rs C
    kCAFChannelLayoutTag_AudioUnit_6_0  = (139<<16) | 6,    // L R Ls Rs C Cs
    kCAFChannelLayoutTag_AudioUnit_7_0  = (140<<16) | 7,    // L R Ls Rs C Rls Rrs
    kCAFChannelLayoutTag_AudioUnit_5_1  = kCAFChannelLayoutTag_MPEG_5_1_A,
                                                             // L R C LFE Ls Rs
    kCAFChannelLayoutTag_AudioUnit_6_1  = kCAFChannelLayoutTag_MPEG_6_1_A,
                                                            // L R C LFE Ls Rs Cs
    kCAFChannelLayoutTag_AudioUnit_7_1  = kCAFChannelLayoutTag_MPEG_7_1_C,
                                                       // L R C LFE Ls Rs Rls Rrs

// These layouts are used for AAC Encoding within the MPEG-4 Specification
    kCAFChannelLayoutTag_AAC_Quadraphonic   = kCAFChannelLayoutTag_Quadraphonic,
                                                              // L R Ls Rs
    kCAFChannelLayoutTag_AAC_4_0= kCAFChannelLayoutTag_MPEG_4_0_B,  // C L R Cs
    kCAFChannelLayoutTag_AAC_5_0= kCAFChannelLayoutTag_MPEG_5_0_D,  // C L R Ls Rs
    kCAFChannelLayoutTag_AAC_5_1= kCAFChannelLayoutTag_MPEG_5_1_D,  // C L R Ls Rs Lfe
    kCAFChannelLayoutTag_AAC_6_0= (141<<16) | 6,            // C L R Ls Rs Cs
    kCAFChannelLayoutTag_AAC_6_1= (142<<16) | 7,            // C L R Ls Rs Cs Lfe
    kCAFChannelLayoutTag_AAC_7_0= (143<<16) | 7,            // C L R Ls Rs Rls Rrs
    kCAFChannelLayoutTag_AAC_7_1= kCAFChannelLayoutTag_MPEG_7_1_B,
                                                            // C Lc Rc L R Ls Rs Lfe
    kCAFChannelLayoutTag_AAC_Octagonal  = (144<<16) | 8,    // C L R Ls Rs Rls Rrs Cs


    kCAFChannelLayoutTag_TMH_10_2_std   = (145<<16) | 16,
                           // L R C Vhc Lsd Rsd Ls Rs Vhl Vhr Lw Rw Csd Cs LFE1 LFE2
```

```
    kCAFChannelLayoutTag_TMH_10_2_full  = (146<<16) | 21,
                                            // TMH_10_2_std plus: Lc Rc HI VI Haptic

    kCAFChannelLayoutTag_RESERVED_DO_NOT_USE= (147<<16)
};
```

Channel Description

If the channel layout tag is set to `kCAFChannelLayoutTag_UseChannelDescriptions`, there is no standard description for the ordering or use of channels in the file; channel descriptions are used instead. In this case, the number of channel descriptions (`mNumberChannelDescriptions`) must equal the number of channels contained in the file. Following the `mNumberChannelDescriptions` field is an array of channel descriptions, one for each channel, as specified by the `CAFChannelDescription` structure:

```
struct CAFChannelDescription
{
    UInt 32             mChannelLabel;
    UInt32              mChannelFlags;
    Float32             mCoordinates[3];
};
```

`mChannelLabel`
: A label that describes the role of the channel. In common cases, such as "Left" or "Right," role implies location. In such cases, `mChannelFlags` and `mCoordinates` can be set to `0`. Refer to "Label Codes for Channel Layouts" (page 34).

`mChannelFlags`
: Flags that indicate how to interpret the data in the `mCoordinates` field. Refer to "Channel Flags for Channel Layouts" (page 36). If the audio channel does not require this information, set this field to `0`.

`mCoordinates`
: A set of three coordinates that specify the placement of the sound source for the channel in three dimensions, according to the `mChannelFlags` information. If the audio channel does not require this information, set this field to `0`.

The number of channel descriptions in this chunk's data section must match the number of channels specified in the `mChannelsPerFrame` field of the Audio Description chunk. In addition, the order of the channel descriptions must correspond to the order of the channels in the Audio Data chunk. See "Audio Description Chunk" (page 17) and "Audio Data Chunk" (page 24).

You can use the optional Information chunk ("Information Chunk" (page 52)) to supply user-presentable names for particular channel layouts. However, if there is any conflict between the channel assignments in the Information chunk and those in the Channel Layout chunk, the Channel Layout chunk always takes precedence.

Label Codes for Channel Layouts

Label Codes indicate the role of a channel. CAF files specify this information in this chunk's `mChannelLabel` field.

The following list includes most channel layouts in common use. Due to differences in channel labeling by various industry groups, there may be overlap or duplication. In every case, use the label that most clearly describes the role of the audio channel.

```
enum
{
    kCAFChannelLabel_Unknown    = 0xFFFFFFFF,   // unknown role or unspecified
                                                // other use for channel
    kCAFChannelLabel_Unused     = 0,            // channel is present, but
                                                // has no intended role or destination
    kCAFChannelLabel_UseCoordinates         = 100,  // channel is described
                                                // solely by the mCoordinates fields

    kCAFChannelLabel_Left                   = 1,
    kCAFChannelLabel_Right                  = 2,
    kCAFChannelLabel_Center                 = 3,
    kCAFChannelLabel_LFEScreen              = 4,
    kCAFChannelLabel_LeftSurround           = 5,    // WAVE (.wav files): "Back Left"
    kCAFChannelLabel_RightSurround          = 6,    // WAVE: "Back Right"
    kCAFChannelLabel_LeftCenter             = 7,
    kCAFChannelLabel_RightCenter            = 8,
    kCAFChannelLabel_CenterSurround         = 9,    // WAVE: "Back Center or
                                                    // plain "Rear Surround"
    kCAFChannelLabel_LeftSurroundDirect     = 10,   // WAVE: "Side Left"
    kCAFChannelLabel_RightSurroundDirect    = 11,   // WAVE: "Side Right"
    kCAFChannelLabel_TopCenterSurround      = 12,
    kCAFChannelLabel_VerticalHeightLeft     = 13,   // WAVE: "Top Front Left"
    kCAFChannelLabel_VerticalHeightCenter   = 14,   // WAVE: "Top Front Center"
    kCAFChannelLabel_VerticalHeightRight    = 15,   // WAVE: "Top Front Right"
    kCAFChannelLabel_TopBackLeft            = 16,
    kCAFChannelLabel_TopBackCenter          = 17,
    kCAFChannelLabel_TopBackRight           = 18,

    kCAFChannelLabel_RearSurroundLeft       = 33,
    kCAFChannelLabel_RearSurroundRight      = 34,
    kCAFChannelLabel_LeftWide               = 35,
    kCAFChannelLabel_RightWide              = 36,
    kCAFChannelLabel_LFE2                   = 37,
    kCAFChannelLabel_LeftTotal              = 38,   // matrix encoded 4 channels
    kCAFChannelLabel_RightTotal             = 39,   // matrix encoded 4 channels
    kCAFChannelLabel_HearingImpaired        = 40,
    kCAFChannelLabel_Narration              = 41,
    kCAFChannelLabel_Mono                   = 42,
    kCAFChannelLabel_DialogCentricMix       = 43,

    kCAFChannelLabel_CenterSurroundDirect   = 44,   // back center, non diffuse
    // first order ambisonic channels
    kCAFChannelLabel_Ambisonic_W            = 200,
    kCAFChannelLabel_Ambisonic_X            = 201,
    kCAFChannelLabel_Ambisonic_Y            = 202,
    kCAFChannelLabel_Ambisonic_Z            = 203,

    // Mid/Side Recording
    kCAFChannelLabel_MS_Mid                 = 204,
    kCAFChannelLabel_MS_Side                = 205,

    // X-Y Recording
    kCAFChannelLabel_XY_X                   = 206,
    kCAFChannelLabel_XY_Y                   = 207,

    // other
    kCAFChannelLabel_HeadphonesLeft         = 301,
```

```
    kCAFChannelLabel_HeadphonesRight        = 302,
    kCAFChannelLabel_ClickTrack             = 304,
    kCAFChannelLabel_ForeignLanguage        = 305
};
```

Channel Flags for Channel Layouts

Channel Flags specify whether a channel layout uses spherical or rectangular coordinates, and whether distances are absolute or relative. CAF files specify this information in this chunk's `mChannelFlags` field.

Here are the CAF conventions for rectangular coordinates:

- Negative is left, and positive is right.
- Negative is back, and positive is front.
- Negative is below ground level, `0` is ground level, and positive is above ground level.

In CAF files, spherical coordinates are measured in degrees. Here are the CAF conventions for spherical coordinates:

- `0` is front center, positive is right, negative is left.
- `+90` is zenith, `0` is horizontal, `-90` is nadir.

These constants are used in the `mChannelFlags` field of the Channel Layout chunk:

```
enum
{
    kCAFChannelFlags_AllOff                     = 0,
    kCAFChannelFlags_RectangularCoordinates     = (1<<0),
    kCAFChannelFlags_SphericalCoordinates       = (1<<1),
    kCAFChannelFlags_Meters                     = (1<<2)
};
```

`kCAFChannelFlags_AllOff`
: No flags are set.

`kCAFChannelFlags_RectangularCoordinates`
: The channel is specified by the cartesian coordinates of the speaker position. This flag is mutually exclusive with `kCAFChannelFlags_SphericalCoordinates`.

`kCAFChannelFlags_SphericalCoordinates`
: The channel is specified by the spherical coordinates of the speaker position. This flag is mutually exclusive with `kCAFChannelFlags_RectangularCoordinates`.

`kCAFChannelFlags_Meters`
: A flag that indicates whether the units are absolute or relative. Set to indicate the units are in meters, clear to indicate the units are relative to the unit cube or unit sphere. For relative units, the listener is assumed to be at the center of the cube or sphere and the maximum radius of the sphere or the distance from the center to the midpoint of the side of the cube is 1.

If the channel description provides no coordinate information, then the `mChannelFlags` field is set to `0`.

Supplementary Data

Some audio formats require specific information in addition to the data in the Audio Description and Audio Data chunks ("Required Chunks" (page 16)). You use the Magic Cookie chunk for this purpose. Similarly, some chunks refer to strings stored in a separate chunk, the Strings chunk.

Magic Cookie Chunk

The Magic Cookie chunk contains supplementary ("magic cookie") data required by certain audio data formats, such as MPEG-4 AAC, for decoding of the audio data. If the audio data format contained in a CAF file requires magic cookie data, the file must have this chunk.

Magic Cookie Chunk Header

Table 3-14 shows the values for the fields in the Magic Cookie chunk header.

Table 2-14 Magic Cookie chunk header fields

Field	Value
`mChunkType`	`'kuki'`
`mChunkSize`	Must always be valid

Magic Cookie Chunk Data Section

The structure of a Magic Cookie chunk's data section is defined by the audio data format it applies to. For example, a CAF file containing MPEG-4 AAC data should have a Magic Cookie chunk containing an elementary stream descriptor. This is the data contained in the 'esds' atom in an MPEG-4 file (and is often referred to as the ESDS) for a given AAC audio track.

Strings Chunk

The optional Strings chunk contains any number of textual strings, along with an index for accessing them. These strings serve as labels for other chunks, such as Marker or Region chunks.

Strings Chunk Header

Table 3-15 shows the values for the fields in the Strings chunk header.

Table 2-15 Strings chunk header fields

Field	Value
`mChunkType`	`'strg'`

Field	Value
mChunkSize	Must always be valid

The Strings chunk header can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data.

Strings Chunk Data Section

The `CAFStrings` structure describes the data section for the Strings chunk.

```
struct CAFStrings
{
    UInt32      mNumEntries;
    CAFStringID mStringsIDs[kVariableLengthArray];
    UInt8       mStrings[kVariableLengthArray];
};
```

`mNumEntries`
The number of strings in the `mStrings` field.

`mStringsIDs`
A lookup table of string IDs for each of the strings in the `mStrings` field. You access strings by using the associated ID. It is recommended that you do not use `0` for an ID.

`mStrings`
An array of null-terminated UTF8-encoded text strings.

String ID

The `CAFStringID` structure describes a string ID, used for accessing a string.

```
struct CAFStringID {
    UInt32  mStringID;
    SInt64  mStringStartByteOffset;
};
typedef struct CAFStringID CAFStringID;
```

`mStringID`
The identifier for the string, allowing applications and other chunks in the file to refer to the string.

`mStringStartByteOffset`
The offset, in bytes, for the start of the string, counting from the first byte after the last `mStringsIDs` entry. The first string has an offset value of `0`.

Marker and Region Chunks

You can add individual markers, marked regions, or both to a CAF file. Marker and Region chunks share some data types, described in the following section. In addition, both can use Strings chunks ("Strings Chunk" (page 37)) to contain text annotations.

Markers and region markers can include timestamps that you can use to correlate the marked point in the audio stream with an external event. For example, you can use a timestamp to correlate a sound in an audio file with a video frame in a movie file. SMPTE (Society of Motion Picture and Television Engineers, pronounced "simptee") time stamps and timecode types are used for this purpose. See "SMPTE Timecode Types" (page 41) and "SMPTE Timestamps" (page 42) for more information on SMPTE time.

Marker Data Types

The data types in this section are used by both the Marker chunk and the Region chunk.

Marker Descriptions

The `CAFMarker` structure defines a marker.

```
struct CAFMarker
{
    UInt32          mType;
    Float64         mFramePosition;
    UInt32          mMarkerID;
    CAF_SMPTE_Time  mSMPTETime;
    UInt32          mChannel;
}
typedef struct CAFMarker CAFMarker;
```

`mType`
: The type of the marker, designated by one of the codes in the Marker Types enumeration. See "Marker Types" (page 39).

`mFramePosition`
: The location of the marker in the file. The location is specified as a frame number, counting from 0 for the first frame in the file.

`mMarkerID`
: The location in the string table (see "Strings Chunk" (page 37)) of a unique ID for the marker description, set by the application. You then use this ID to refer to the marker. It is recommended that you do not use `0` for an ID.

`mSMPTETime`
: A SMPTE timestamp for the marker. You can use this field to relate a marker in the CAF file to a time in another file, such as a video file. Mark the SMPTE timestamp as invalid if you do not need this feature. To indicate that a marker's SMPTE timestamp is not valid, set all of its bytes to `0xFF`. See "SMPTE Timestamps" (page 42).

`mChannel`
: The channel, by number, to which the marker description applies. This number corresponds to the sequence in which the data for the channels is ordered in the frame. The first channel is numbered 1. Set this field to 0 to indicate that the marker applies to all channels.

Marker Types

The following enumeration lists the supported marker types for CAF files. Use these codes in the `mType` field of each marker description (see the `CAFMarker` structure, above in this section).

```
enum {
```

```
    kCAFMarkerType_Generic              = 0,
    kCAFMarkerType_ProgramStart         = 'pbeg',
    kCAFMarkerType_ProgramEnd           = 'pend',
    kCAFMarkerType_TrackStart           = 'tbeg',
    kCAFMarkerType_TrackEnd             = 'tend',
    kCAFMarkerType_Index                = 'indx',
    kCAFMarkerType_RegionStart          = 'rbeg',
    kCAFMarkerType_RegionEnd            = 'rend',
    kCAFMarkerType_RegionSyncPoint      = 'rsyc',
    kCAFMarkerType_SelectionStart       = 'sbeg',
    kCAFMarkerType_SelectionEnd         = 'send',
    kCAFMarkerType_EditSourceBegin      = 'cbeg',
    kCAFMarkerType_EditSourceEnd        = 'cend',
    kCAFMarkerType_EditDestinationBegin = 'dbeg',
    kCAFMarkerType_EditDestinationEnd   = 'dend',
    kCAFMarkerType_SustainLoopStart     = 'slbg',
    kCAFMarkerType_SustainLoopEnd       = 'slen',
    kCAFMarkerType_ReleaseLoopStart     = 'rlbg',
    kCAFMarkerType_ReleaseLoopEnd       = 'rlen'
};
```

`kCAFMarkerType_Generic`
: Generic marker.

`kCAFMarkerType_ProgramStart`
: Start-of-program marker; used to delineate the start of a CD or other playlist.

`kCAFMarkerType_ProgramEnd`
: End-of-program marker; used to delineate the end of a CD.

`kCAFMarkerType_TrackStart`
: Start-of-track marker; used to delineate the start of a track for a CD.

`kCAFMarkerType_TrackEnd`
: End-of-track marker; used to delineate the end of a track for a CD.

`kCAFMarkerType_Index`
: Index marker for a Red Book compliant index.

`kCAFMarkerType_RegionStart`
: Start-of-region marker. See "Region Chunk" (page 44).

`kCAFMarkerType_RegionEnd`
: End-of-region marker. See "Region Chunk" (page 44).

`kCAFMarkerType_RegionSyncPoint`
: Region synchronization point marker; used to synchronize a point in (or external to) a region with an event, such as beat in the music.

`kCAFMarkerType_SelectionStart`
: Start-of-selection marker, for user selection of a portion of a displayed waveform.

`kCAFMarkerType_SelectionEnd`
: End-of-selection marker, for user selection of a portion of a displayed waveform.

`kCAFMarkerType_EditSourceBegin`
: Beginning-of-source marker for a copy or move operation.

`kCAFMarkerType_EditSourceEnd`
: End-of-source marker for a copy or move operation.

`kCAFMarkerType_EditDestinationBegin`
: Beginning-of-destination marker for a copy or move operation.

`kCAFMarkerType_EditDestinationEnd`
: End-of-destination marker for a copy or move operation.

`kCAFMarkerType_SustainLoopStart`
: Start-of-sustain marker for a sustain loop.

`kCAFMarkerType_SustainLoopEnd`
: End-of-sustain marker for a sustain loop.

`kCAFMarkerType_ReleaseLoopStart`
: Start-of-release marker for a sustain loop.

`kCAFMarkerType_ReleaseLoopEnd`
: End-of-release marker for a sustain loop.

SMPTE Timecode Types

The following enumeration lists the supported SMPTE timecode types for CAF files. Timecode types are used by the Marker and Region chunks to synchronize the data in a CAF file with the data in a video file (see "Marker Chunk Data Section" (page 43) and "Region Chunk Data Section" (page 44)).

```
enum
{
    kCAF_SMPTE_TimeTypeNone     = 0,
    kCAF_SMPTE_TimeType24       = 1,
    kCAF_SMPTE_TimeType25       = 2,
    kCAF_SMPTE_TimeType30Drop   = 3,
    kCAF_SMPTE_TimeType30       = 4,
    kCAF_SMPTE_TimeType2997     = 5,
    kCAF_SMPTE_TimeType2997Drop = 6,
    kCAF_SMPTE_TimeType60       = 7,
    kCAF_SMPTE_TimeType5994     = 8
};
```

`kCAF_SMPTE_TimeTypeNone`
: No timecode type is assigned. Use this value if you are not specifying a SMPTE time in the marker.

`kCAF_SMPTE_TimeType24`
: 24 video frames per second—standard for 16mm and 35mm film.

`kCAF_SMPTE_TimeType25`
: 25 video frames per second—standard for PAL and SECAM video.

`kCAF_SMPTE_TimeType30Drop`
: 30 video frames per second, with video-frame-number counts adjusted to ensure that the timecode matches elapsed clock time.

`kCAF_SMPTE_TimeType30`
: 30 video frames per second.

`kCAF_SMPTE_TimeType2997`
: 29.97 video frames per second—standard for NTSC video.

`kCAF_SMPTE_TimeType2997Drop`
: 29.97 video frames per second—standard for NTSC video—with video-frame-number counts adjusted to ensure that the timecode matches elapsed clock time.

`kCAF_SMPTE_TimeType60`
: 60 video frames per second.

`kCAF_SMPTE_TimeType5994`
: 59.94 video frames per second.

SMPTE Timestamps

Each marker may contain a SMPTE timestamp in its `mSMPTETime` field that you can use to associate a marker with an external SMPTE time (see "Marker Descriptions" (page 39))—for example, to synchronize the audio data with a video file.

The `CAF_SMPTE_Time` structure describes the format for indicating timestamps in a CAF file.

```
struct CAF_SMPTE_Time
{
    SInt8   mHours;
    SInt8   mMinutes;
    SInt8   mSeconds;
    SInt8   mFrames;
    UInt32  mSubFrameSampleOffset;
};
typedef struct CAF_SMPTE_Time CAF_SMPTE_Time;
```

`mHours`
: The number of hours for the timestamp.

`mMinutes`
: The number of minutes for the timestamp.

`mSeconds`
: The number of seconds for the timestamp.

`mFrames`
: The number of video frames for the timestamp. Use the SMPTE timecode type ("SMPTE Timecode Types" (page 41)) to determine the number of video frames per second.

`mSubFrameSampleOffset`
: An audio sample offset to the HH:MM:SS:FF time stamp. You can use this field to position the marker somewhere within the time span represented by a video frame, if necessary. The `mSampleRate` field (see "Audio Description Chunk Data Section" (page 17)) specifies the number of audio frames per second for this CAF file.

To indicate an unused SMPTE timestamp, set every byte in the `CAF_SMPTE_Time` structure to `0xFF`. When a CAF file does not specify a SMPTE timecode type (see "SMPTE Timecode Types" (page 41)), all marker description timestamps must be set as invalid.

Marker Chunk

You can use the optional Marker chunk to contain any number of marker descriptions, each of which marks a particular sample location in the file.

Marker descriptions may also use a timing convention known as SMPTE (Society of Motion Picture and Television Engineers) timecode. For more information on this convention, see http://www.smpte.org/.

Marker Chunk Header

Table 3-16 shows the values for the fields in the Marker chunk header.

Table 2-16 Marker chunk header fields

Field	Value
`mChunkType`	`'mark'`
`mChunkSize`	Must always be valid

The Marker chunk header can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data.

Marker Chunk Data Section

The Marker chunk data section has two informational fields followed by a list of marker descriptions. The `CAFMarkerChunk` structure describes the data section for this chunk.

```
struct CAFMarkerChunk
{
    UInt32      mSMPTE_TimeType;
    UInt32      mNumberMarkers;
    CAFMarker   mMarkers[kVariableLengthArray];
}
```

`mSMPTE_TimeType`

The type of SMPTE timecode used for the markers. For the types available, see "SMPTE Timecode Types" (page 41). You should use a SMPTE timestamp only if you need to synchronize a marker in the CAF file with an external event, such as a point in a video file. To indicate that the markers in the file do not have valid SMPTE timestamps, set this field to `0`.

If this field has a nonzero value, you should interpret marker description timestamps according to the specified timecode type. Individual marker descriptions can still have invalid (`0xFF`) SMPTE timestamps.

A CAF file can contain markers with no regions (see "Region Chunk" (page 44)), regions with no Marker chunk, or both a Marker chunk and a Region chunk. For this reason, the Marker and Region chunks both include an `mSMPTE_TimeType` field. In typical use, if both chunks are present, the value in both fields is identical.

`mNumberMarkers`

The total number of marker descriptions in this chunk, starting immediately after this field and continuing until the end of this chunk. This number must always be valid.

`mMarkers`

The marker descriptions. See "Marker Descriptions" (page 39). The Marker chunk data section contains 0 or more marker descriptions.

Region Chunk

You can use the optional Region chunk to contain any number of region descriptions. Each region description includes starting and ending marker descriptions that delineate a span of sample frames in the audio data. See "Marker Descriptions" (page 39) for more information about markers. A region description can contain more than two markers, with the purpose of the additional markers being application defined.

Region Chunk Header

Table 3-17 shows the values for the fields in the Region chunk header.

Table 2-17 Region chunk header fields

Field	Value
`mChunkType`	`'regn'`
`mChunkSize`	Must always be valid

The Region chunk header can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data.

Region Chunk Data Section

The Region chunk data section has two informational fields followed by a list of region descriptions. The `CAFRegionChunk` structure describes the data section for this chunk.

```
struct CAFRegionChunk
{
    UInt32      mSMPTE_TimeType;
    UInt32      mNumberRegions;
    CAFRegion   mRegions[kVariableLengthArray];
}
typedef struct CAFRegionChunk CAFRegionChunk;
```

`mSMPTE_TimeType`

The type of SMPTE timecode used for the markers. For the types available, see "SMPTE Timecode Types" (page 41). You should use a SMPTE timestamp only if you need to synchronize a region in the CAF file with a region in another file, such as a video file. To indicate that the markers in the file do not have valid timestamps, set this field to `0`.

If this field has a nonzero value, you should interpret marker description timestamps according to the specified timecode type. Individual marker descriptions can still have invalid (`0xFF`) SMPTE timestamps.

A CAF file can contain regions with no Marker chunk (see "Marker Chunk" (page 42)), a Marker chunk with no regions, or both a Marker chunk and a Region chunk. For this reason, both the Marker and Region chunks include an `mSMPTE_TimeType` field. In typical use, if both chunks are present, the value in both fields is identical.

`mNumberRegions`

The number of region descriptions in the data section.

`mRegions`
: The region descriptions.

Region Description

The Region chunk data section contains 0 or more region descriptions. The `CAFRegion` structure defines a region description. Region descriptions are referred to by the Instrument chunk; see "Instrument Chunk Data Section" (page 46).

```
struct CAFRegion
{
    UInt32      mRegionID;
    UInt32      mFlags;
    UInt32      mNumberMarkers;
    CAFMarker   mMarkers[kVariableLengthArray];
};
typedef struct CAFRegion CAFRegion;
```

`mRegionID`
: A unique ID for the region description, set by the application. You then use this ID to refer to the region. It is recommended that you do not use `0` for a region ID.

`mFlags`
: A flag providing some information about the purpose of the region. See "Region Flags" (page 45) for possible values.

`mNumberMarkers`
: The total number of marker descriptions in this region description. This number must always be valid.

`mMarkers`
: The marker descriptions for this region.

Region Flags

Each region description includes a set of flags, defined by the following enumeration:

```
enum {
    kCAFRegionFlag_LoopEnable   = 1,
    kCAFRegionFlag_PlayForward  = 2,
    kCAFRegionFlag_PlayBackward = 4
};
```

`kCAFRegionFlag_LoopEnable`
: If this flag is set, the audio data delineated by this region should be played as a loop. If this flag is set, then one or both of the `PlayForward` and `PlayBackward` flags must also be set.

`kCAFRegionFlag_PlayForward`
: If this flag is set, the loop should be played forward. If both this flag and the `PlayBackward` flag are set, then the loop should be played alternately forward and backward.

`kCAFRegionFlag_PlayBackward`
: If this flag is set, the loop should be played backward. If both this flag and the `PlayForward` flag are set, then the loop should be played alternately forward and backward.

Music Metadata

Two chunk types, the Instrument chunk and the MIDI chunk, provide information of importance to the interpretation of certain music data.

Instrument Chunk

The optional Instrument chunk can be used to describe the audio data in a CAF file in terms relevant to samplers or to other digital audio processing applications. For example, a file or a portion of a file can be described as a MIDI instrument. (For more information about MIDI and MIDI instruments, go to http://www.midi.org/.)There can be any number of Instrument chunks in a CAF file, each specifying a portion of the file.

Instrument Chunk Header

Table 3-18 shows the values for the fields in the Instrument chunk header.

Table 2-18 Instrument chunk header fields

Field	Value
`mChunkType`	`'inst'`
`mChunkSize`	Must always be valid

Instrument Chunk Data Section

The Instrument chunk data section has informational fields and a list of region descriptions. The `CAFInstrumentChunk` structure describes the data section for this chunk.

```
struct CAFInstrumentChunk
{
    Float32 mBaseNote;
    UInt8   mMIDILowNote;
    UInt8   mMIDIHighNote;
    UInt8   mMIDILowVelocity;
    UInt8   mMIDIHighVelocity;
    Float32 mdBGain;
    UInt32  mStartRegionID;
    UInt32  mSustainRegionID;
    UInt32  mReleaseRegionID;
    UInt32  mInstrumentID;
};
typedef struct CAFInstrumentChunk CAFInstrumentChunk;
```

`mBaseNote`

The MIDI note number, and fractional pitch, for the base note of the MIDI instrument. The integer portion of this field indicates the base note, in the integer range `0` to `127`, where a value of `60` represents middle C and each integer is a step on a standard piano keyboard (for example, 61 is C#

above middle C). The fractional part of the field specifies the fractional pitch; for example, 60.5 is a pitch halfway between notes 60 and 61.

`mMIDILowNote`

The lowest note for the region, in the integer range `0` to `127`, where a value of `60` represents middle C (following the MIDI convention). This value represents the suggested lowest note on a keyboard for playback of this instrument definition. The sound data should be played if the instrument is requested to play a note between `mMIDILowNote` and `mMIDIHighNote`, inclusive. The `mBaseNote` value must be within this range.

`mMIDIHighNote`

The highest note for the region when used as a MIDI instrument, in the integer range `0` to `127`, where a value of `60` represents middle C. See the discussions of the `mBaseNote` and `mMIDILowNote` fields for more information.

`mMIDILowVelocity`

The lowest MIDI velocity for playing the region , in the integer range `0` to `127`.

`mMIDIHighVelocity`

The highest MIDI velocity for playing the region, in the integer range `0` to `127`.

`mdBGain`

The gain, in decibels, for playing the region. A value of `0` represents unity gain. Use negative numbers to indicate a decrease in gain.

`mStartRegionID`

The ID of the region (see"Region Description" (page 45)) that defines the portion of the file to use as the "start" stage for a MIDI instrument. A lack of a valid region ID in this field indicates that there is no start stage. It is recommended that you do not assign an ID of `0` to any region description, so that you can use `0` in this and the following fields to indicate the lack of a region ID.

`mSustainRegionID`

The ID of the region (in the Region chunk) that defines the portion of the file to use as the "sustain" stage for a MIDI instrument. A lack of a valid region ID in this field indicates that there is no sustain stage.

`mReleaseRegionID`

The ID of the region (in the Region chunk) that defines the portion of the file to use as the "release" stage for a MIDI instrument. A lack of a valid region ID in this field indicates that there is no release stage.

`mInstrumentID`

The ID of the string (in the Strings chunk, "Strings Chunk" (page 37)) that specifies the name of the instrument. A lack of a valid string ID in this field means that no name is specified. It is recommended that you do not assign an ID of `0` to any string description, so that you can use `0` in this field to indicate the lack of a string ID.

MIDI Chunk

You can use the optional MIDI chunk to contain MIDI data using the standard MIDI file format. It can be used to store metadata about the audio in the file's Data chunk, or even a MIDI representation of that audio. For information on the MIDI standard, see http://www.midi.org.

You should consider information in this chunk to supersede conflicting information in the Information chunk ("Information Chunk" (page 52)). For example, both the Information chunk and the MIDI chunk may specify key signature and tempo. In that case, the MIDI chunk values should override the values in the Information chunk.

MIDI Chunk Header

Table 3-19 shows the values for the fields in the MIDI chunk header.

Table 2-19 MIDI chunk header fields

Field	Value
`mChunkType`	`'midi'`
`mChunkSize`	Must always be valid

The MIDI chunk header must specify the true size of the valid data in the data section.

MIDI Chunk Data Section

The data section of a MIDI Chunk can be used to hold anything that can be described by a standard MIDI file, such as:

- Tempo information
- Key signature
- Time signature
- MIDI representation of the audio data; for example, MIDI note numbers

Audio Editor Support

You can use the Overview chunk to hold sample descriptions of the audio data for displaying the data for the user, and the Peak chunk to hold information about peak amplitudes.

Overview Chunk

You can use the optional Overview chunk to hold sample descriptions that you can use to draw a graphical view of the audio data in a CAF file. A CAF file can include multiple Overview chunks to represent the audio at multiple graphical resolutions.

Overview Chunk Header

Table 3-20 shows the values for the fields in the Overview chunk header.

Table 2-20 Overview chunk header fields

Field	Value
`mChunkType`	`'ovvw'`

Field	Value
mChunkSize	Must always be valid

The Overview chunk header must specify the true size of the valid data in the data section.

Overview Chunk Data Section

The Overview chunk data section has two informational fields followed by a list of sample descriptions. The CAFOverview structure describes the data section for this chunk.

```
struct CAFOverview
{
    UInt32              mEditCount;
    UInt32              mNumFramesPerOVWSample;
    CAFOverviewSample   mData[kVariableLengthArray];
};
typedef struct CAFOverview CAFOverview;
```

`mEditCount`
: The modification count of the Overview Chunk data section. When you create an Overview chunk, you should set the `mEditCount` field to the value of the `mEditCount` field of the CAF file's Audio Data chunk. You can then check whether an overview is still valid by comparing the edit counts. If they don't match, you should regenerate the overview.

`mNumFramesPerOVWSample`
: The number of frames of audio data that are represented by a single overview sample.

`mData`
: An array of overview samples. For the `mNumFramesPerOVWSample` frames of audio in the Audio Data chunk, you must store one sample per channel in this field. The sequence of channels should be the same as in the Audio Data chunk.

Overview Sample

The Overview chunk data section contains overview samples, described by the `CAFOverviewSample` structure.

```
struct CAFOverviewSample
{
    SInt16  mMinValue;
    SInt16  mMaxValue;
};
```

`mMinValue`
: The minimum value for the sample, listed as a big-endian, 16-bit signed integer.

`mMaxValue`
: The maximum value for the sample, listed as a big-endian, 16-bit signed integer.

Peak Chunk

You can use the optional Peak chunk to describe the peak amplitude present in each channel of a CAF file and to indicate in which frame the peak occurs for each channel.

Peak Chunk Header

Table 3-21 shows the values for the fields in the Peak chunk header.

Table 2-21 Peak chunk header fields

Field	Value
`mChunkType`	`'peak'`
`mChunkSize`	Must always be valid

The Peak chunk uses a Peak structure to describe each peak (see "Peak Structure" (page 51)). The size of a Peak chunk's data section, to be placed in the `mChunkSize` field of the header, depends on the number of channels in the file as follows:

```
mChunkSize = sizeof(CAFPositionPeak) * numChannelsInFile + sizeof(UInt32);
```

The `sizeof(UInt32)` argument represents the data section's `mEditCount` field. The number of channels in the file, represented by the `numChannelsInFile` argument, is specified in the `mChannelsPerFrame` field of the Audio Description chunk.

Peak Chunk Data Section

The Peak chunk data section contains a field for edit count, followed by a list of Peak structures. The `CAFPeakChunk` structure describes the data section for the Peak chunk.

```
struct CAFPeakChunk
{
    UInt32          mEditCount;
    CAFPositionPeak mPeaks[kVariableLengthArray];
};
typedef struct CAFPeakChunk CAFPeakChunk;
```

`mEditCount`

The modification status of the Peak Chunk data section. When you create a Peak chunk, set the `mEditCount` field to the value of the `mEditCount` field of the CAF file's Audio Data chunk. You can then check whether the peak data is still valid by comparing the edit counts. If they don't match, the peak information must be regenerated.

`mPeaks`

An array of Peak structures, one for each channel of audio data contained in the file. See "Peak Structure" (page 51).

The number of channels in the file is specified in the `mChannelsPerFrame` field of the Audio Description chunk ("Audio Description Chunk" (page 17)).

Peak Structure

The Peak chunk data section contains one Peak structure for each channel, defined as follows:

```
struct CAFPositionPeak
{
    Float32 mValue;
    UInt64  mFrameNumber;
};
```

`mValue`
: The signed maximum absolute amplitude in a channel, normalized to a floating-point value in the interval [{–1.0, +1.0}].

`mFrameNumber`
: The frame number where the peak occurs. The first frame in a CAF file is `0`.

Annotations

You can add text strings to the CAF file to provide information about the audio data (in the Information chunk) and to indicate what editing has been done on the file (in the Edit Comments chunk).

Edit Comments Chunk

You can use the optional Edit Comments chunk to carry time-stamped, human-readable comments that coincide with edits to the audio data in a CAF file.

Edit Comments Chunk Header

Table 3-22 shows the values for the fields in the Edit Comments chunk header.

Table 2-22 Edit Comments chunk header fields

Field	Value
`mChunkType`	`'edct'`
`mChunkSize`	Must always be valid

The Edit Comments chunk header can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data.

Edit Comments Chunk Data Section

The data section for this chunk contains a field describing the number of entries, followed by a list of edit comments. The `CAFCommentStringsChunk` structure describes the data section for the Edit Comments chunk.

```
struct CAFCommentStringsChunk
{
    UInt32  mNumEntries;
    mStrings[kVariableLengthArray];     // variable length
};
```

`mNumEntries`
The number of edit comments in the data section.

`mStrings`
A list of edit comments. See "Edit Comment" (page 52).

Edit Comment

The `editComment` structure describes an edit comment.

```
struct editCommment {
    UInt8   mKey[kVariableLengthArray];
    UInt8   mValue[kVariableLengthArray];
}
```

`mKey`
A null-terminated, time-of-day string that conforms to ISO-8601. All times are based on UTC (Coordinated Universal Time). See "Time Of Day Data Format" (page 59).

`mValue`
A null-terminated UTF8 string.

Information Chunk

You can use the optional Information chunk to contain any number of human-readable text strings. Each string is accessed through a standard or application-defined key.

You should consider information in this chunk to be secondary when the same information appears in other chunks. For example, both the Information chunk and the MIDI chunk ("MIDI Chunk" (page 47)) may specify key signature and tempo. In that case, the MIDI chunk values overrides the values in the Information chunk.

Information Chunk Header

Table 3-23 shows the values for the fields in the Information chunk header.

Table 2-23 Information chunk header fields

Field	Value
`mChunkType`	`'info'`
`mChunkSize`	Must always be valid

The Information chunk header can specify a data section size that is larger than the chunk's current meaningful content in order to reserve room for additional data.

Information Chunk Data Section

The `CAFStringsChunk` structure describes the data section for the Information chunk.

```
struct CAFStringsChunk
{
    UInt32  mNumEntries;
    mStrings[kVariableLengthArray];             // variable length
};
```

`mNumEntries`
The number of information strings in the chunk. Must always be valid.

`mStrings`
A variable-length keyed array of information entries. See "Information Entries" (page 53).

CAF includes some conventions for the Information chunk's key-value pairs.

- Apple reserves keys that are all lowercase (see "Information Entry Keys" (page 53)). Application-defined keys should include at least one uppercase character.
- For any key that ends with `' date'` (that is, the space character followed by the word `'date'`—for example, `'recorded date'`), the value must be a time-of-day string. See "Time Of Day Data Format" (page 59).
- Using a `'.'` (period) character as the first character of a key means that the key-value pair is not to be displayed. This allows you to store private information that should be preserved by other applications but not displayed to a user.

Information Entries

The `CAFInformation` structure describes an information entry.

```
struct CAFInformation
{
    UInt8   mKey[kVariableLengthArray];
    UInt8   mValue[kVariableLengthArray];
};
```

`mKey`
A null-terminated UTF8 string. See "Information Entry Keys" (page 53).

`mValue`
A null-terminated UTF8 string.

Information Entry Keys

Apple reserves keys that are all lowercase. Application-defined keys should contain at least one uppercase character. Each key can be used only once. You can specify multiple values for a single key by separating the values with commas. The following are the standard keys for the Information chunk:

`tempo`
The base tempo of the audio data in beats per minute.

`key signature`
: The key signature for the audio in the file. In the `mValue` field, the note is capitalized with values from `A` to `G`. Lowercase `m` indicates a minor key. Lowercase `b` indicates a flat key. The `#` symbol indicates a sharp key.

  Examples: `'C'`, `'Cm'`, `'C#'`, `'Cb'`.

`time signature`
: The time signature for the audio in the file.

  Examples: `'4/4'`, `'6/8'`.

`artist`
: The name of the performance artist for the audio in the file.

  Example: `'Able Baker,Charlie Delta'`

`album`
: The name of the album that the audio in the file is a part of.

`track number`
: The track number, within the album, for the audio in the file.

`year`
: The year of publication for the audio in the file.

`composer`
: The name of the composer for the audio in the file.

`lyricist`
: The name of the lyricist for the audio in the file.

`genre`
: The name of the genre for the audio in the file.

`title`
: The title or name of the audio in the file. Can be different from the filename.

`recorded date`
: A timestamp for the recording in the file. See "Time Of Day Data Format" (page 59).

`comments`
: Freeform comments about the audio in the file.

`copyright`
: Copyright information for the audio in the file.

  Example: `'Copyright © 2004 The CoolBandName. All Rights Reserved'`

`source encoder`
: Description of the encoding algorithm, if any, used for the audio in the file.

  Example: `'My AAC Encoder v4.2'`

`encoding application`
: Description of the encoding application, if any, used for the audio in the file.

  Example: `'My App v1.0'`

`nominal bit rate`
: Description of the bit rate used for the audio in the file.

  Example: `'128 kbits'`

`channel layout`
: Description of the channel layout for the file.

  Examples: `'stereo'`, `'5.1 Surround'`, `'10.2 Surround'`

Identifier

CAF files can include a Unique Material Identifier chunk to uniquely identify the audio content.

Unique Material Identifier Chunk

You can use the optional Unique Material Identifier chunk to uniquely identify the audio contained in a CAF file. There can be at most one UMID chunk within a file.

The data in this chunk conforms to the standard SMPTE 330M-2004 specification for unique material identifiers. See http://www.smpte.org/standards/.

The European Broadcasting Union (EBU) provides guidelines for use of UMIDs in broadcast production. CAF files should adhere to these guidelines. See http://www.ebu.ch/CMSimages/en/tec_text_d92-2001_tcm6-4721.pdf.

Unique Material Identifier Chunk Header

Table 3-24 shows the values for the fields in the Unique Material Identifier chunk header.

Table 2-24 Unique Material Identifier chunk header fields

Field	Value
`mChunkType`	`'umid'`
`mChunkSize`	64 (`sizeof(CAFUMIDChunk)`); Must always be valid

Unique Material Identifier Chunk Data Section

The `CAFUMIDChunk` structure describes the UMID chunk's data section.

```
struct CAFUMIDChunk
{
    UInt8 mBytes[64];
};
typedef struct CAFUMIDChunk CAFUMIDChunk;
```

`mBytes`

The UMID for the file. The first 32 bytes constitute the "Basic" UMID and include four pieces of information: instance number, flag indicating copy or original, material number, and description of device that recorded the original material.

The second 32 bytes constitute the so-called "Source Pack" section for the UMID, which includes three additional pieces of information: timestamp of recording, geographic coordinates of recording, and ownership information.

The size of a UMID chunk's data section is exactly 64 bytes. If a CAF file has only a "Basic" UMID, the remaining 32 bytes in the data section should be set to `0`.

For more information, refer to the UMID specification, SMPTE 330M-2004, available from http://www.smpte.org/standards/.

Extending the CAF Specification

You can define your own chunk type to extend the CAF file specification. For this purpose, this specification includes the User-Defined chunk type, which you can use to provide a unique universal identifier for your custom chunk.

When parsing a CAF file, you should ignore any chunk with a UUID that you do not recognize.

User-Defined Chunk

If you define your own, custom chunk, you can use the User-Defined chunk type to assign a universally unique ID to the chunk.

User-Defined Chunk Header

Table 3-25 shows the values for the fields in the User-Defined chunk header.

Table 2-25 CAF header field values for User-Defined Chunk

Field	Value
`mChunkType`	`'uuid'`
`mChunkSize`	The size of the data section plus 16 bytes for the UUID. Must always be valid

In addition to the standard fields, the header of a custom chunk includes a universal identifier, as shown in the `CAF_UUID_ChunkHeader` structure.

```
struct CAF_UUID_ChunkHeader
{
    CAFChunkHeader  mHeader;
    UInt8           mUUID[16];
};
CAF_UUID_ChunkHeader CAF_UUID_ChunkHeader;
```

mHeader
: The standard CAF header with the values in Table 3-25.

mUUID
: A unique universal identifier (UUID), based on the ISO 14496-1 specification for UUID identifiers, available from http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList.

User-Defined Chunk Data Section

Any data following the chunk header is defined by the custom chunk type. If the UUID chunk has dependencies on the edit count of the Audio Data chunk, then the edit count should be stored after the `mUUID` field.

Extra Space

In many chunk types, you can specify a larger chunk size than is currently needed for data in order to reserve additional space within the chunk. To reserve extra space in the CAF file as a whole, use a Free chunk.

Free Chunk

The optional Free chunk is for reserving space, or providing padding, in a CAF file. The contents of the Free chunk data section have no significance and should be ignored.

Free Chunk Header

Table 3-26 shows the values for the fields in the Free chunk header.

Table 2-26 Free chunk header fields

Field	Value
`mChunkType`	`'free'`
`mChunkSize`	Must always be valid

Set `mChunkSize` to the size of the data section you are using for reserved space.

Free Chunk Data Section

You should ignore the contents of the Free chunk data section.

Time Of Day Data Format

The time-of-day fields used by the Edit Comments chunk ("Edit Comments Chunk" (page 51)) and the Information chunk ("Information Chunk" (page 52)) are based on the ISO 8601 specification for time-of-day strings, available from http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList. Time-of-day symbols are shown in Table A-1.

Table A-1 Symbols used in time-of-day formats

Symbol	Meaning
YYYY	4-digit year
MM	2-digit month (01=January, etc.)
DD	2-digit day of month (01 through 31)
'T'	separator between date and time fragments
hh	2-digit hour (00 through 23) (am/pm not allowed)
mm	2-digit minute (00 through 59)
ss	2-digit second (00 through 59)

Some example formats are:

Year
YYYY (`2005`)

Year and month
YYYY-MM (`2005-07`)

Complete date
YYYY-MM-DD (`2005-07-16`)

Complete date plus hours, minutes and seconds
YYYY-MM-DDThh:mm:ss (`2005-07-16T19:20:30`)

Note that the CAF specification's use of this standard does not include fractional seconds.

Document Revision History

This table describes the changes to *Apple Core Audio Format Specification 1.0*.

Date	Notes
2006-03-08	Corrected the description of the mStringStartByteOffset field.
2005-06-04	New document that specifies the Apple Core Audio Format (CAF) for audio files.

Document Revision History