This chapter describes the Sound Manager, the part of the Macintosh system software that controls the production and manipulation of sounds on Macintosh computers. You can use the Sound Manager to create a wide variety of sounds and to manipulate sounds in many ways. The Sound Manager is also used by other parts of the Macintosh system software that produce sounds, such as the Speech Manager and QuickTime.

To use this chapter, you should already be familiar with the information in the chapter "Introduction to Sound on the Macintosh" earlier in this book, especially with the portions of that chapter that describe the Macintosh sound architecture and the routines related to sound output. That chapter shows how your application can play a sound resource or a sound file synchronously (that is, with other processing suspended while the sound plays).

You should read this chapter if you need a greater degree of control over sound output than the routines described in that introductory chapter provide. For example, if you want to play sounds asynchronously or to exercise very fine control over the process of sound production, this chapter contains information you need.

This chapter begins by describing the capabilities of the Sound Manager and the role of sound commands and sound channels in producing sound. Then it explains how you can use the Sound Manager to

- create and manage sound channels
- obtain information about available sound features and sound channels
- play notes and other sounds at various frequencies and volumes
- play one or more sounds asynchronously
- parse sound resources and sound files to obtain information about them
- compress and expand sound data
- use double buffers to bypass the normal play-from-disk routines

You're not likely to use all of these capabilities in a single application. In general, you should read the section "About the Sound Manager" and then turn to the parts of the section "Using the Sound Manager" that describe the features you want to use in your application. The section "Sound Storage Formats" beginning on page 2-73 explains in detail the format of sound resources and sound files. You can find a complete reference to the Sound Manager data structures and routines in the section "Sound Manager Reference" beginning on page 2-89.

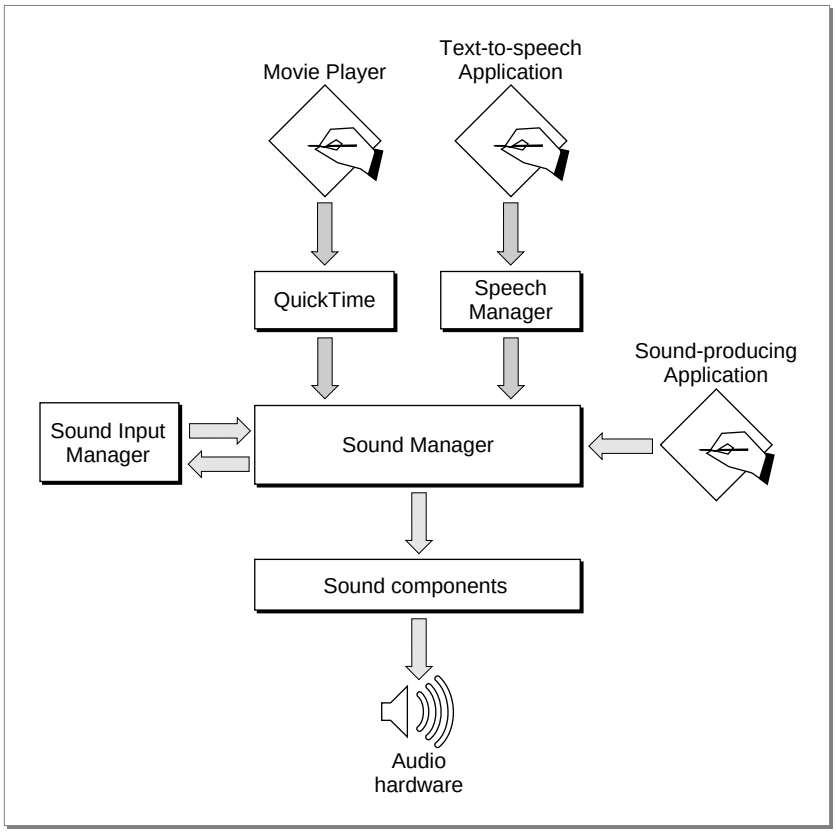
IMPORTANT

This chapter describes the capabilities and programming interfaces of version 3.0 of the Sound Manager. See the chapter "Introduction to Sound on the Macintosh" for some information on how version 3.0 differs from earlier versions. The capabilities and performance of version 3.0 are significantly better than those of all previous Sound Manager versions, even though their programming interfaces are largely identical. This chapter occasionally warns you about techniques or routines that cannot be used in versions prior to 3.0, but it does not provide an exhaustive comparison of all available versions. ▲

About the Sound Manager

The Sound Manager is a collection of routines that your application can use to create sound without a knowledge of or dependence on the actual sound-producing hardware available on any particular Macintosh computer. More generally, the Sound Manager is responsible for managing all sound production on Macintosh computers. Other parts of the Macintosh system software that need to create or modify sounds use the Sound Manager to do so. Figure 2-1 shows the position of the Sound Manager in relation to sound-producing applications and to other parts of the system software, such as the Speech Manager and QuickTime.

Figure 2-1 The position of the Sound Manager



The Sound Manager was first introduced in system software version 6.0 and has been significantly enhanced since that time. Prior to system software version 6.0, applications could create sounds using the Sound Driver.

IMPORTANT
To ensure compatibility across all models of Macintosh computers, you should always use the Sound Manager rather than the Sound Driver, which is no longer documented or supported by Apple Computer, Inc. The Sound Manager is simpler and much more powerful than the Sound Driver. Moreover, Sound Driver code might not work on some Macintosh computers. ▲

This section describes the three basic ways of defining sounds, namely using wave-table data, square-wave data, or sampled-sound data. Usually, you'll use sampled data to define the sounds you want to create, because sampled data provides the greatest flexibility and variety of sounds. You might use wave-table or square-wave data for very simple sounds. For instance, the Simple Beep alert sound is defined using square-wave data. Most other alert sounds are defined using sampled-sound data.

This section also describes sound commands and sound channels, which you need to know about to be able to do anything more complex than play sound resources or files synchronously using high-level Sound Manager routines.

Sound Data

The Sound Manager can play sounds defined using one of three kinds of sound data:

- square-wave data
- wave-table data
- sampled-sound data

This section provides a brief description of each of these kinds of audio data and introduces some of the concepts that are used in the remainder of this chapter. A complete description of the nature and format of audio data is beyond the scope of this book. There are, however, numerous books available that provide complete discussions of digital audio data.

Square-Wave Data

Square-wave data is the simplest kind of audio data supported by the Sound Manager. You can use square-wave data to generate a sound based on a square wave. Your application can use square-wave data to play a simple sequence of sounds in which each sound is described completely by three factors: its frequency or pitch, its amplitude (or volume), and its duration.

The **frequency** of a sound is the number of cycles per second (or hertz) of the sound wave. Usually, you specify a sound's frequency by a MIDI value. **MIDI note values** correspond to frequencies for musical notes, such as middle C, which is defined to have a MIDI value of 60, which on Macintosh computers is equivalent to 261.625 hertz.

Pitch is a lister's subjective interpretation of the sound's frequency. The terms *frequency* and *pitch* are used interchangeably in this chapter.

A sound's **duration** is the length of time a sound takes to play. In the Sound Manager, durations are usually specified in half-milliseconds.

The **amplitude** of a sound is the loudness at which it is being played. Two sounds played at the same amplitude might not necessarily sound equally loud. For example, one sound could be played at a lower volume (which the user may set with the Sound control panel). Or, a sampled sound of a fleeting whisper might sound softer than a sampled sound of continuous gunfire, even if your application plays them at the same amplitude.

Note
Amplitude is traditionally considered to be the height of a sound wave, so that two sounds with the same amplitude would always sound equally loud. However, the Sound Manager considers amplitude to be the adjustment to be made to an existing sound wave. A sound played at maximum amplitude still might sound soft if the wave amplitude is small. ◆

A sound's **timbre** is its clarity. A sound with a low timbre is very clear; a sound with a high timbre is buzzing. Only sounds defined using square-wave data have timbres.

Wave-Table Data

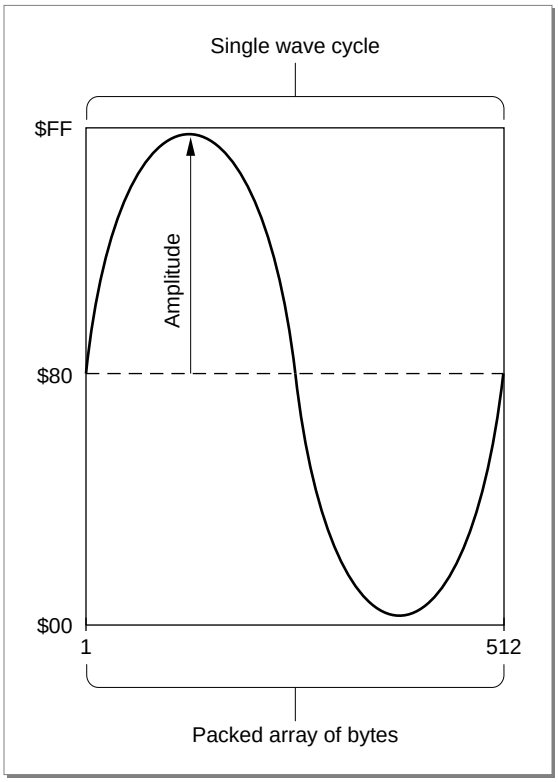
To produce more complex sounds than are possible using square-wave data, your applications can use **wave-table data.** As the name indicates, wave-table data is based on a description of a single wave cycle. This cycle is called a wave table and is represented as an array of bytes that describe the timbre (or tone) of a sound at any point in the cycle.

Your application can use any number of bytes to represent the wave, but 512 is the recommended number because the Sound Manager resizes a wave table to 512 bytes if the table is not exactly that long. Your application can compute the wave table at run time or load it from a resource.

A **wave table** is a sequence of wave amplitudes measured at fixed intervals. For instance, a sine wave can be converted into a wave table by taking the value of the wave's amplitude at every $^{1}/_{512}$ interval of the wave (see Figure 2-2).

A wave table is represented as a packed array of bytes. Each byte contains a value in the range $00–$FF. These values are interpreted as offset values, where $80 represents an amplitude of 0. The largest negative amplitude is $00 and the largest positive amplitude is $FF. When playing a wave-table description of a sound, the Sound Manager loops through the wave table for the duration of the sound.

Figure 2-2 A graph of a wave table



Sampled-Sound Data

You can use **sampled-sound data** to play back sounds that have been digitally recorded (that is, **sampled sounds**) as well as sounds that are computed, possibly at run time. Sampled sounds are the most widely used of all the available sound types primarily because it is relatively easy to generate a sampled sound and because sampled-sound data can describe a wide variety of sounds. Sampled sounds are typically used to play back prerecorded sounds such as speech or special sound effects.

You can use the Sound Manager to store sampled sounds in one of two ways, either as resources of type `'snd '` or as AIFF or AIFF-C format files. The structure of resources of type `'snd '` is given in "Sound Resources" on page 2-74, and the structure of AIFF and AIFF-C files is given in "Sound Files" on page 2-81. If you simply want to play short prerecorded sampled sounds, you should probably include the sound data in `'snd '` resources. If you want to allow the user to transfer recorded sound data from one application to another (or from one operating system to another), you should probably store the sound data in an AIFF or AIFF-C file. In certain cases, you must store sampled sounds in files and not in resources. For example, a sampled sound might be too large to be stored in a resource.

Regardless of how you store a sampled sound, you can use Sound Manager routines to play that sound. If you choose to store sampled sounds in files of type AIFF or AIFF-C,

you can play those sounds by calling the `SndStartFilePlay` function, introduced in the chapter "Introduction to Sound on the Macintosh" in this book. If you store sampled sounds in resources, your application can play those sounds by passing the Sound Manager function `SndPlay` a handle to a resource of type `'snd '` that contains a sampled sound header. (The `SndStartFilePlay` function can also play `'snd '` resources directly from disk, but this is not recommended.)

There are three types of sampled-sound headers: the standard sound header, the extended sound header, and the compressed sound header. The **sound header** contains information about the sample (such as the original sampling rate, the length of the sample, and so forth), together with an indication of where the sample data is to be found. The **sampled sound header** can reference only buffers of monophonic, 8-bit sound. The **extended sound header** can be used for 8- or 16-bit stereo sound data as well as monophonic sound data. The **compressed sound header** can be used to describe compressed sound data, whether monophonic or stereo. Data can be stored in a buffer separate from the sound resource or as part of the sound resource as the last field of the sound header.

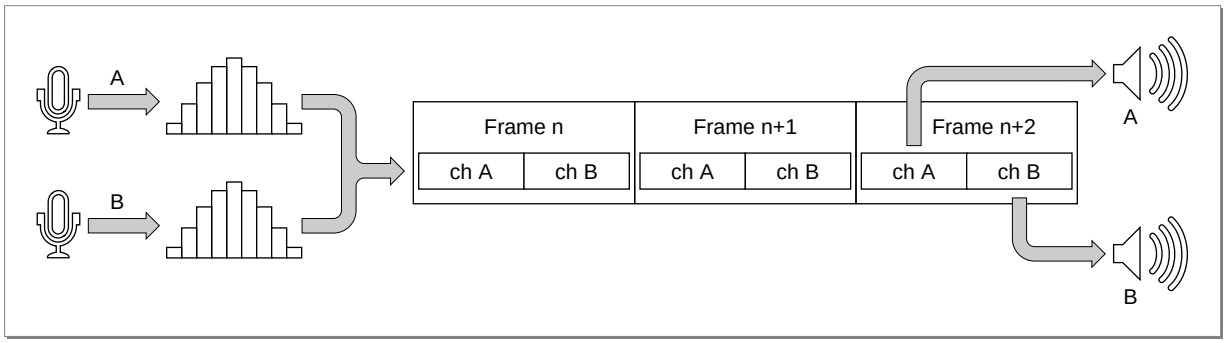
Note
The terminology *sampled sound header* can be confusing because in most cases the sound header (and hence the `'snd '` resource) contains the sound data as well as information describing the data. Also, do not confuse sampled sound headers with sound resource headers. Sampled sound headers contain information about sampled-sound data, but sound resource headers contain information on the format of an entire sound resource. ◆

You can play a sampled sound at its original rate or play it at some other rate to change its pitch. Once you install a sampled sound header into a channel, you can play it at varying rates to provide a number of pitches. In this way, you can use a sampled sound as a **voice** or **instrument** to play a series of sounds.

Sampled-sound data is made up of a series of **sample frames,** which are stored contiguously in order of increasing time. For noncompressed sound data, each sample frame contains one or more sample points. For compressed sound data, each sample frame contains one or more packets.

For multichannel sounds, a sample frame is an interleaved set of sample points or packets. (For monophonic sounds, a sample frame is just a single sample point or a single packet.) The sample points within a sample frame are interleaved by channel number. For example, the sound data for a stereo, noncompressed sound is illustrated in Figure 2-3.

Figure 2-3 Interleaving stereo sample points



Each **sample point** of noncompressed sound data in a sample frame is, for sound files, a linear, two's complement value, and, for sound resources, a binary offset value. Sample points are from 1 to 32 bits wide. The size is usually 8 bits, but a different size can be specified in the `sampleSize` field of the extended sound header (for sound resources) or in the `sampleSize` field of the Common Chunk (for sound files). Each sample point is stored in an integral number of contiguous bytes. Sample points that are from 1 to 8 bits wide are stored in 1 byte, sample points that are from 9 to 16 bits wide are stored in 2 bytes, and so forth. When the width of a sample point is less than a multiple of 8 bits, the sample point data is left aligned (using a shift-left instruction), and the low-order bits at the right end are set to 0.

For example, for 8-bit noncompressed sound data stored in a sound resource, each sample point is similar to a value in a wave-table description. These values are interpreted as offset values, where $80 represents an amplitude of 0. The value $00 is the most negative amplitude, and $FF is the largest positive amplitude.

Each **packet** of 3:1 compressed sound data is 2 bytes; a packet of 6:1 compressed sound is 1 byte. These byte sizes are defined in bits by the constants `threeToOnePacketSize` and `sixToOnePacketSize`, respectively.

Sound Commands

The Sound Manager provides routines that allow you to create and dispose of sound channels. These routines allow you to manipulate sound channels, but they do not directly produce any sounds. To actually produce sounds, you need to issue sound commands. A **sound command** is an instruction to produce sound, modify sound, or otherwise assist in the overall process of sound production. For example, the `ampCmd` sound command changes the amplitude (or volume) of a sound.

You can issue sound commands in several ways. You can send sound commands one at a time into a sound channel by repeatedly calling the `SndDoCommand` function. The commands are held in a queue and processed in a first-in, first-out order. Alternatively, you can bypass a sound queue altogether by calling the `SndDoImmediate` function. You can also issue sound commands by calling the function `SndPlay` and specifying a sound resource of type `'snd '` that contains the sound commands you want to issue. A sound

resource can contain any number of sound commands. As a result, you might be able to accomplish all sound-related activity simply by creating sound resources and calling `SndPlay` in your application. See "Sound Resources" on page 2-74 for details on the format of an `'snd '` resource.

Generally speaking, no matter how sound commands are issued, they are all eventually sent to the Sound Manager, which interprets the commands and plays the sound on the available audio hardware. The Sound Manager provides a rich set of sound commands. The structure of a sound command is defined by the `SndCommand` data type:

```
TYPE SndCommand =
PACKED RECORD
   cmd:        Integer;    {command number}
   param1:     Integer;    {first parameter}
   param2:     LongInt;    {second parameter}
END;
```

Commands are always 8 bytes in length. The first 2 bytes are the command number, and the next 6 make up the command's options. The format of the last 6 bytes depends on the command in use, although typically those 6 bytes are interpreted as an integer followed by a long integer. For example, an application can install a wave table into a sound channel by using `SndDoCommand` with a sound command whose `cmd` field is the `waveTableCmd` constant. In that case, the `param1` field specifies the length of the wave table, and the `param2` field is a pointer to the wave-table data itself. Other sound commands may interpret the 6 parameter bytes differently or may not use them at all.

The sound commands available to your application are defined by constants.

```
CONST
   nullCmd           = 0;     {do nothing}
   quietCmd          = 3;     {stop a sound that is playing}
   flushCmd          = 4;     {flush a sound channel}
   reInitCmd         = 5;     {reinitialize a sound channel}
   waitCmd           = 10;    {suspend processing in a channel}
   pauseCmd          = 11;    {pause processing in a channel}
   resumeCmd         = 12;    {resume processing in a channel}
   callBackCmd       = 13;    {execute a callback procedure}
   syncCmd           = 14;    {synchronize channels}
   availableCmd      = 24;    {see if initialization options are supported}
   versionCmd        = 25;    {determine version}
   totalLoadCmd      = 26;    {report total CPU load}
   loadCmd           = 27;    {report CPU load for a new channel}
   freqDurationCmd   = 40;    {play a note for a duration}
   restCmd           = 41;    {rest a channel for a duration}
   freqCmd           = 42;    {change the pitch of a sound}
   ampCmd            = 43;    {change the amplitude of a sound}
   timbreCmd         = 44;    {change the timbre of a sound}
```

```
getAmpCmd         = 45;    {get the amplitude of a sound}
volumeCmd         = 46;    {set volume}
getVolumeCmd      = 47;    {get volume}
waveTableCmd      = 60;    {install a wave table as a voice}
soundCmd          = 80;    {install a sampled sound as a voice}
bufferCmd         = 81;    {play a sampled sound}
rateCmd           = 82;    {set the pitch of a sampled sound}
getRateCmd        = 85;    {get the pitch of a sampled sound}
```

For details on individual sound commands, see the relevant sections in "Using the Sound Manager" beginning on page 2-17. Also see "Sound Command Numbers" beginning on page 2-92 for a complete summary of the available sound commands, their parameters, and their uses.

Sound Channels

A **sound channel** is a queue of sound commands that is managed by the Sound Manager, together with other information about the sounds to be played in that channel. The commands placed into the channel might originate from an application or from the Sound Manager itself. The commands in the queue are passed one by one, in a first-in, first-out (FIFO) manner, to the Sound Manager for interpretation and processing.

The Sound Manager uses the `SndChannel` data type to define a sound channel.

```
TYPE SndChannel =
PACKED RECORD
   nextChan:      SndChannelPtr; {pointer to next channel}
   firstMod:      Ptr;           {used internally}
   callBack:      ProcPtr;       {pointer to callback procedure}
   userInfo:      LongInt;       {free for application's use}
   wait:          LongInt;       {used internally}
   cmdInProgress: SndCommand;    {used internally}
   flags:         Integer;       {used internally}
   qLength:       Integer;       {used internally}
   qHead:         Integer;       {used internally}
   qTail:         Integer;       {used internally}
   queue:         ARRAY[0..stdQLength-1] OF SndCommand;
END;
```

Most of the fields of the **sound channel record** are used internally by the Sound Manager, and you should not access or change them. However, your application is free to use the `userInfo` field to store any information that you wish to associate with a sound channel. For example, you might store a handle to an application-defined record that contains information about how your application is using the channel.

Some applications do not need to worry about creating or disposing of sound channels because the high-level Sound Manager routines take care of these automatically.

However, if you wish to customize sound output or play sounds asynchronously, you must create your own sound channels (with the `SndNewChannel` function).

The enhanced Sound Manager included in system software versions 6.0.7 and later provides the ability to have multiple channels of sampled sound produce output on the Macintosh audio hardware concurrently. (Previous versions of the Sound Manager could play only a single channel of sampled sound at a time.) This allows a layering of sound that can bring a touch of reality to a simulation or presentation and permits applications to incorporate synthesized speech output with any other kind of Macintosh-generated sound. Sound Manager version 3.0 extended this capability to allow multiple channels of any kind of sound data to play simultaneously.

Your application can open several channels of sound for concurrent output on the available audio hardware. Similarly, multiple applications can each open channels of sound. The number and quality of concurrent channels of sound are limited only by the abilities of the machine, particularly by the speed of the CPU. Different Macintosh computers have different CPU clock speeds and execute instructions at quite different rates. This means that some machines can manage more channels of sound and produce higher-quality sound than other machines. For example, a Macintosh Quadra might be able to support several channels of high-quality stereo sound without significant impact on other processing, whereas a Macintosh Plus might be able to support only a single channel of monophonic sound before other processing slows significantly.

The Sound Manager currently supports multiple channels of sound only on machines equipped with an Apple Sound Chip or equivalent hardware. To maintain maximum compatibility between machines for your applications, you should always check the operating environment to make sure that the ability to play multiple channels of sampled sound is present before attempting to do so. A technique for determining whether your application can play multiple channels of sound is described in “Testing for Multichannel Sound and Play-From-Disk Capabilities” on page 2-35.

Sound Compression and Expansion

One minute of monophonic sound recorded with the fidelity you would expect from a commercial compact disc occupies about 5.3 MB of disk space. One minute of sound digitized by the current low-fidelity digitizing peripherals for Macintosh computers occupies more than 1 MB of disk space. Even one minute of telephone-quality speech takes up more than half of a megabyte on a disk. Despite the increased capacities of mass-storage devices, disk space can be a problem if your application incorporates large amounts of sampled sound. The space problem is particularly acute for multimedia applications. Because a large portion of the space occupied by a multimedia application is likely to be taken up by sound data, the complexity and richness of the application’s sound component are limited.

To help remedy this problem, the Sound Manager includes a set of routines known collectively as **Macintosh Audio Compression and Expansion (MACE).** MACE enables you to provide more audio information in a given amount of storage space by allowing you to compress sound data and then expand it for playback. These enhancements are based entirely in software and require no specialized hardware.

The audio compression and expansion features allow you to enhance your application by including more audio data. MACE also relieves some distribution problems by reducing the number of disks required for shipping an application that relies heavily on sound. MACE has made some kinds of applications, such as talking dictionaries and foreign language-instruction software, more feasible than before.

MACE adds three main kinds of capabilities to those already present in the Sound Manager: audio data compression, real-time expansion and playback of compressed audio data, and buffered expansion and playback of compressed audio data.

- **Compression.** The Sound Manager can compress a buffer of digital audio data either in the original buffer or in a separate buffer. If a segment of audio data is too large to fit into a single buffer, your application can make repeated calls to the compression routine.
- **Real-time expansion playback.** The Sound Manager can expand compressed audio data contained in a small internal buffer and play it back at the same time. Because the audio data expansion and playback occur at the same time, there is more of a strain on the CPU when using this method of sound expansion rather than buffered expansion.
- **Buffered expansion.** The Sound Manager can expand a specified buffer of compressed audio data and store the result in a separate buffer. The expanded buffer can then be played back using other Sound Manager routines with minimal processor overhead during playback. Applications that require screen updates or user interaction during playback (such as animation or multimedia applications) should use buffered expansion.

MACE provides audio data compression and expansion capabilities in ratios of either 3:1 or 6:1 for all currently supported Macintosh models, from the Macintosh Plus forward. The principal tradeoff when using MACE is that the expanded audio data suffers a loss of fidelity in comparison to the original data. A small amount of noise is introduced into a 3:1 compressed sound when it is expanded and played back, and a greater amount of noise for the 6:1 ratio. The 3:1 buffer-to-buffer compression and expansion option is well suited for high-fidelity sounds. The 6:1 buffer-to-buffer compression and expansion option provides greater compression at the expense of lower-fidelity results and is recommended for voice data only. This technique reduces the frequency bandwidth of the audio signal by a factor of two to achieve the higher compression ratio.

MACE allows for the compression of both monophonic and stereo sounds. However, some Macintosh computer models (such as the Macintosh Plus and Macintosh SE) cannot expand stereo sounds.

Note
With Sound Manager versions prior to 3.0, some Macintosh computers play only the right channel of stereo `'snd '` data through the internal speaker. Certain Macintosh II models can play only a single channel through the internal speaker. Sound Manager version 3.0 removes both of these limitations. ◆

Existing applications that use the Sound Manager's `SndPlay` function to play digitized audio signals can play compressed audio signals without modification or recompilation.

The MACE routines assume that each original sample consists of 8-bit sound in binary offset format. The compression techniques do not, however, depend on a particular **sample rate** (the rate at which samples are recorded). Table 2-1 shows some common sample rates, expressed both as hertz and as unsigned fixed-point values.

Table 2-1 Sample rates

Rate (Hz)	Sample rate value (Fixed)
44100.00000	$AC440000
22254.54545	$56EE8BA3
22050.00000	$56EE8BA3
11127.27273	$2B7745D1
11025.00000	$2B110000
7418.1818	$1CFA2E8B
5563.6363	$15BBA2E8

The Sound Manager defines constants for the most common sample rates:

```
CONST
   rate44khz         = $AC440000;   {44100.00000 in fixed-point}
   rate22khz         = $56EE8BA3;   {22254.54545 in fixed-point}
   rate22050hz       = $56220000;   {22050.00000 in fixed-point}
   rate11khz         = $2B7745D1;   {11127.27273 in fixed-point}
   rate11025hz       = $2B110000;   {11025.00000 in fixed-point}
```

The compression techniques produce their best quality output when the sample rate is the same as the output rate of the sound hardware of the machine playing the audio data. The output rate used in most current Macintosh computers is 22.254 kilohertz (hereafter referred to as the 22 kHz rate). Because of speed limitations, the Macintosh Plus and Macintosh SE cannot perform sample-rate conversion during expansion playback. On those machines, all sounds are played back at a 22 kHz rate. To provide consistent quality in sounds that might be played on different machines, you should record all sounds at a 22 kHz sample rate.

The MACE algorithms are optimized to provide the best sound quality possible through the internal speaker in real time. However, the user who employs high-quality speakers might notice a high-frequency hiss for some sounds compressed at the 3:1 ratio. This hiss results from a design tradeoff between maintaining real-time operation on the Macintosh Plus and preserving as much frequency bandwidth of the signal as possible. If you think that your output might be played on high-quality speakers, you might want to filter out the hiss before compression by passing the audio output through an equalizer that removes frequencies above 10 kHz. When you use the 6:1 compression and expansion ratio, your frequency response is cut in half. For example, when you use the 22 kHz

sample rate, the highest frequency possible would normally be 11 kHz; however, after compressing and expanding the data at the 6:1 ratio, the highest frequency you could get would be only 5.5 kHz.

Note
The Sound Manager uses compressions and decompression components (codecs) to handle the MACE capabilities. You can provide custom codecs to use other compression and decompression algorithms. See the chapter "Sound Components" in this book for information on developing audio codecs. ◆

Using the Sound Manager

The Sound Manager provides a wide variety of methods for creating sound and manipulating audio data on Macintosh computers. Usually, your application needs to use only a few of the many routines or sound commands that are available.

The Sound Manager routines can be divided into high-level routines and low-level routines. The high-level routines (like `SndPlay` and `SysBeep`) give you the ability to produce very complex audio output at very little programming expense. The majority of applications interact with the Sound Manager using these high-level routines, which allow you to play sounds without knowing anything about the structure of sound commands or sampled-sound data. You can let the high-level routines automatically allocate channels, or, for increased control, you can allocate your own sound channels.

Applications that have more sophisticated sound capabilities use the low-level routines (like `SndDoCommand` and `SndDoImmediate`) to send sound commands to sound channels. For example, your application might send a sound command to alter the amplitude of a sound that is playing (or is about to play).

Finally, a few very specialized applications use the Sound Manager's low-level sound playback routines, which allow fine-tuning of the algorithms the Sound Manager uses to manage the double buffering of sound for its play-from-disk routines.

In general, you should use the highest-level routines capable of producing the kind of sound you want. Many applications can simply play sounds stored in resources or files and do not need to customize the sounds or continue with other processing while those sounds are playing. In such cases, you can use the high-level Sound Manager routines, as illustrated in the chapter "Introduction to Sound on the Macintosh" in this book. If, however, you need to be able to exercise very fine control over sound output or to play sounds asynchronously, you must manage your own sound channels. See "Managing Sound Channels" on page 2-19 to learn how you can use the Sound Manager to

- allocate and dispose of sound channels manually by using the `SndNewChannel` and `SndDisposeChannel` functions
- manipulate sound that is playing (for example, by sending the `ampCmd` command to a sound channel to change the amplitude of sound playing)

- stop sounds and flush sound channels by using the `quietCmd` and `flushCmd` commands
- pause and restart sound channels by using the `pauseCmd` and `resumeCmd` commands
- synchronize sound channels by using the `syncCmd` command

As you've learned, the capabilities of the Sound Manager vary greatly from one Macintosh computer to another, depending on which version of the Sound Manager is available on a particular computer and on what audio hardware is available. To create sounds effectively on all computers, you might need to obtain information about the available sound features. "Obtaining Sound-Related Information" on page 2-32 explains how you can

- use the `Gestalt` function to determine which basic sound features are available
- find the version number of the available Sound Manager or of the MACE compression and expansion routines
- determine whether your application can take advantage of multichannel sound and the play-from-disk routines
- obtain information about a single sound channel

Some applications need to be able to play computer-generated tones at different pitches. In addition, some applications need to play waveforms or sampled sounds at different pitches. For example, if you are writing an application that converts musical notes to sound, you might record the sound of a violin playing middle C and then replay the sound at a variety of pitches to simulate a violinist's playing a concerto. The Sound Manager allows you to do this by allocating a sound channel and sending sound commands to it. "Playing Notes" on page 2-41 explains how you can

- play simple sequences of notes by using the `freqCmd` and `freqDurationCmd` commands
- install waveforms or sampled sounds into channels by using the `soundCmd` and `waveTableCmd` commands so that you can play them at different frequencies
- set a sound resource's loop points so that the sound repeats if a `freqCmd` or `freqDurationCmd` command lasts longer than the sound

Although some applications do not need to do other processing while sounds are playing, others do. If your application allocates sound channels itself, it can request that the Sound Manager play sounds asynchronously. By using callback procedures and completion routines, your application can arrange for a sound channel to be disposed when a sound finishes playing. "Playing Sounds Asynchronously" on page 2-46 explains how you can

- play a sound resource asynchronously by defining a callback procedure
- use callback procedures to synchronize sounds you play asynchronously with other actions
- play a sound file asynchronously and pause, restart, or stop such an asynchronous playback

- manage multiple channels of sound to play more than one sound asynchronously at the same time

The high-level Sound Manager routines automatically parse sound resources and sound files to determine the information the Sound Manager needs to play the sounds contained in the resources and files. However, you might need to obtain information about sound resources or sound files for some other reason. Or, you might need to locate a certain part of a sound resource or sound file. For example, to use the `bufferCmd` sound command to play a buffer of sampled sound, you must obtain a pointer to the sound header contained in that buffer. See the section "Parsing Sound Resources and Sound Files" on page 2-56 for information on how to

- parse sound resources containing sampled-sound data to obtain information from the sampled-sound data's sound header
- use the `bufferCmd` command to play sampled-sound data stored within a sound resource
- parse sound files to find a particular chunk and to extract the data from that chunk

High-level Sound Manager routines automatically expand sound data in real time when playing compressed sounds. However, you might need to manually compress or expand sound data at a time when you are not playing sounds. "Compressing and Expanding Sounds" on page 2-66 explains how you can use the Sound Manager's built-in sound compression and expansion routines to compress or expand sounds.

The Sound Manager's high-level play-from-disk routines use highly optimized algorithms to manage the double buffering of data so that the play from disk is continuous and without audible gaps. However, if you wish to bypass the high-level Sound Manager play-from-disk routines, you may define your own double-buffering routines. This might be useful if you need to change the sound data on disk before the Sound Manager can process it. The section "Using Double Buffers" on page 2-68 explains how you can set up your own double buffers and use a doubleback procedure to bypass the normal play-from-disk routines.

Managing Sound Channels

To use most of the low-level Sound Manager routines, you must specify a sound channel that maintains a queue of commands. Also, to take advantage of the full capabilities of the high-level Sound Manager routines, including asynchronous sound play, you must allocate your own sound channels. This section explains how your application can allocate, dispose of, and use its own sound channels.

This section first describes how you can allocate and dispose of sound channels. Then it explains how you can manipulate sounds playing in sound channels, stop sounds playing in sound channels, and pause and restart the execution of sounds in sound channels.

Allocating Sound Channels

Usually, you do not need to worry about allocating memory for sound channels because the `SndNewChannel` function automatically allocates a sound channel record in the application's heap if passed a pointer to a `NIL` sound channel. `SndNewChannel` also internally allocates memory for the sound channel's queue of sound commands. For example, the following lines of code request that the Sound Manager open a new sound channel for playing sampled sounds:

```
mySndChan := NIL;
myErr := SndNewChannel(mySndChan, sampledSynth, 0, NIL);
```

If you are concerned with managing memory yourself, you can allocate your own memory for a sound channel record and pass the address of that memory as the first parameter to `SndNewChannel`. By allocating a sound channel record manually, you not only obtain control over the allocation of the sound channel record, but you can specify the size of the queue of sound commands that the Sound Manager internally allocates. Listing 2-1 illustrates one way to do this.

Listing 2-1 Creating a sound channel

```
FUNCTION MyCreateSndChannel (synth: Integer; initOptions: LongInt;
                             userRoutine: ProcPtr;
                             queueLength: Integer): SndChannelPtr;
VAR
   mySndChan:  SndChannelPtr;    {pointer to a sound channel}
   myErr:      OSErr;
BEGIN
   {Allocate memory for sound channel.}
   mySndChan := SndChannelPtr(NewPtr(Sizeof(SndChannel)));
   IF mySndChan <> NIL THEN
   BEGIN
      mySndChan^.qLength := queueLength;  {set number of commands in queue}
      {Create a new sound channel.}
      myErr := SndNewChannel(mySndChan, synth, initOptions, userRoutine);
      IF myErr <> noErr THEN
      BEGIN                               {couldn't allocate channel}
         DisposePtr(Ptr(mySndChan));      {free memory already allocated}
         mySndChan := NIL;                {return NIL}
      END
      ELSE
         mySndChan^.userInfo := 0;        {reset userInfo field}
   END;
   MyCreateSndChannel := mySndChan;       {return new sound channel}
END;
```

The `MyCreateSndChannel` function defined in Listing 2-1 first allocates memory for a sound channel record and then calls the `SndNewChannel` function to attempt to allocate a channel. Note that `MyCreateSndChannel` checks the result code returned by `SndNewChannel` to determine whether the function was able to allocate a channel. The `SndNewChannel` function might not be able to allocate a channel if there are so many channels open that allocating another would put too much strain on the CPU. Also, `SndNewChannel` might fail if memory is low. (In addition to the memory for a sound channel record that is passed in the first parameter to `SndNewChannel`, the function must internally allocate memory in which to store sound commands.)

If you allocate memory for a sound channel record, you should specify the size of the queue of sound commands by assigning a value to the `qLength` field of the sound channel record you allocate. You can use the constant `stdQLength` to obtain a standard queue of 128 sound commands, or you can provide a value of your own.

```
CONST
   stdQLength           = 128;   {default size of a sound channel}
```

If you know that your application will play only resources containing sampled sound, you might set the `qLength` field to a considerably lower value, because resources created with the `SndRecord` function (described in the chapter "Introduction to Sound on the Macintosh" in this book) contain only one sound command, the `bufferCmd` command, which specifies that a buffer of sound should be played. For example, if your application uses a sound channel only to play a single sampled sound asynchronously, you can set `qLength` to 2, to allow for the `bufferCmd` command and a `callBackCmd` command that your application issues manually, as described in "Playing Sounds Asynchronously" on page 2-46. By using a smaller than standard queue length, your application can conserve memory.

Note
The number of sound commands in a channel should be an integer greater than 0. If you open a channel with a 0-length queue, most of the Sound Manager routines will return a `badChannel` result code. ◆

IMPORTANT
In general, however, you should let the Sound Manager allocate sound channel records for you. The amount of memory you might save by allocating your own is usually negligible. ▲

The second parameter in the `SndNewChannel` function specifies the kind of data you want to play on that channel. You can specify one of the following constants:

```
CONST
   squareWaveSynth   = 1;     {square-wave data}
   waveTableSynth    = 3;     {wave-table data}
   sampledSynth      = 5;     {sampled-sound data}
```

In some versions of system software prior to system software version 7.0 (including system software version 6.0.7), high-level Sound Manager routines do not work properly

with sound resources that specify the sound data type twice. This might happen if a resource specifies that a sound consists of sampled-sound data and an application does the same when creating a sound channel. This might also happen if an application uses the same sound channel to play several sound resources that contain different kinds of sound data. There are several solutions to this problem that you can use if you must maintain compatibility with old versions of system software:

- If your application plays only sampled-sound resources, then you need only ensure that none of the sound resources specifies that it contains sampled-sound data. Then, when you create a sound channel, pass `sampledSynth` as the second parameter to `SndNewChannel` so that the Sound Manager interprets the data in the sound resources correctly. Do not use the `SndPlay` routine.
- If your application must be able to play sampled-sound resources as well as resources that contain square-wave or wave-table data, ensure that all sound resources that your application uses specify their data type. (Sound resources created with the Sound Input Manager automatically specify that they contain sampled-sound data.) Then, when creating a channel in which you plan to play a sound resource, pass 0 as the second parameter to `SndNewChannel`, and then use the channel to play no more than one sound resource.
- If you do not wish to modify your application's sound resources, and your application plays only sampled-sound resources, then you can play sounds with low-level Sound Manager routines, a technique described in "Playing Sounds Using Low-Level Routines" on page 2-61.

Note that this problem does not occur with sound files, because sound files always contain sampled-sound data and thus do not explicitly declare their data type. As a result, when creating a channel in which you plan to play a sound file, pass `sampledSynth` as the second parameter to `SndNewChannel`.

The third parameter in the `SndNewChannel` function specifies the initialization parameters to be associated with the new channel. These are discussed in the following section. The fourth parameter in the `SndNewChannel` function is a pointer to a callback procedure. If your application produces sounds asynchronously or needs to be alerted when a command has completed, you can specify a callback procedure by passing the address of that procedure in the fourth parameter and then by installing a callback procedure into the sound channel. If you pass `NIL` as the fourth parameter, then no callback procedure is associated with the channel. See "Playing Sounds Asynchronously" on page 2-46 for more information on setting up and using callback procedures.

Initializing Sound Channels

When you first create a sound channel with `SndNewChannel`, you can request that the channel have certain characteristics as specified by a sound channel initialization parameter. For example, to indicate that you want to allocate a channel capable of producing stereo sound, you might use the following code:

```
myErr := SndNewChannel(mySndChan, sampledSynth, initStereo, NIL);
```

These are the currently recognized constants for the sound channel initialization parameter.

```
CONST
   initChanLeft      = $0002;    {left stereo channel}
   initChanRight     = $0003;    {right stereo channel}
   waveInitChannel0  = $0004;    {wave-table channel 0}
   waveInitChannel1  = $0005;    {wave-table channel 1}
   waveInitChanne12  = $0006;    {wave-table channel 2}
   waveInitChannel3  = $0007;    {wave-table channel 3}
   initMono          = $0080;    {monophonic channel}
   initStereo        = $00C0;    {stereo channel}
   initMACE3         = $0300;    {3:1 compression}
   initMACE6         = $0400;    {6:1 compression}
   initNoInterp      = $0004;    {no linear interpolation}
   initNoDrop        = $0008;    {no drop-sample conversion}
```

See “Channel Initialization Parameters” beginning on page 2-91 for a complete description of these constants.

Note

Some Macintosh computers play *only* the left channel of stereo sounds out the internal speaker. Other machines (for example, the Macintosh SE/30 and Macintosh IIsi) mix both channels together before sending a signal to the internal speaker. You can use the `Gestalt` function to determine if a particular machine mixes both left and right channels to the internal speaker. All Macintosh computers except the Macintosh SE and the Macintosh Plus, however, play stereo signals out the headphone jack. ◆

The initialization parameters are additive. To initialize a channel for stereo sound with no linear interpolation, simply pass an initialization parameter that is the sum of the desired characteristics, as follows:

```
myErr := SndNewChannel(mySndChan, sampledSynth,
                       initStereo+initNoInterp, NIL);
```

A call to `SndNewChannel` is really only a request that the Sound Manager open a channel having the desired characteristics. It is possible that the parameters requested are not available. In that case, `SndNewChannel` returns a `notEnoughHardwareErr` error. In general, you should pass 0 as the third parameter to `SndNewChannel` unless you know exactly what kind of sound is to be played.

You can alter certain initialization parameters, even while a channel is actively playing a sound, by issuing the `reInitCmd` command. For example, you can change the output channel from left to right, as shown in Listing 2-2.

Listing 2-2 Reinitializing a sound channel

```
VAR
   mySndCmd:          SndCommand;
   mySndChan:         SndChannelPtr;
   myErr:             OSErr;
.
.
.
mySndCmd.cmd := reInitCmd;
mySndCmd.param1 := 0;                          {unused}
mySndCmd.param2 := initChanRight;              {new init parameter}
myErr := SndDoImmediate(mySndChan, mySndCmd);
```

The `reInitCmd` command accepts the `initNoInterp` constant to toggle **linear interpolation** on and off; it should be used with noncompressed sounds only. If an noncompressed sound is playing when you send a `reInitCmd` command with this constant, linear interpolation begins immediately. You can also pass `initMono`, `initChanLeft`, or `initChanRight` to pan to both channels, to the left channel, or to the right channel. This affects only monophonic sounds. The Sound Manager remembers the settings you pass and applies them to all further sounds played on that channel.

Releasing Sound Channels

To dispose of a sound channel that you have allocated with `SndNewChannel`, use the `SndDisposeChannel` function. `SndDisposeChannel` requires two parameters, a pointer to the channel that is to be disposed and a Boolean value that indicates whether the channel should be flushed before disposal. Here's an example:

```
myErr := SndDisposeChannel(mySndChan, TRUE);
```

Because the second parameter is `TRUE`, the Sound Manager sends both a `flushCmd` command and a `quietCmd` command to the sound channel (using `SndDoImmediate`). This removes all commands from the sound channel and stops any sound already in progress. Then the Sound Manager disposes of the channel.

If the second parameter is `FALSE`, the Sound Manager simply queues a `quietCmd` command (using `SndDoCommand`) and waits until `quietCmd` is received by the channel before disposing of the channel. In this case, the `SndDisposeChannel` function does not return until the channel has finished processing commands and the queue is empty.

▲ **WARNING**
If you dispose of a channel currently playing from disk, then your completion routine will still execute, but will receive a pointer to a sound channel that no longer exists. Thus, you should stop a play from disk before disposing of a channel. See "Managing an Asynchronous Play From Disk" on page 2-52 for more information on completion routines. ▲

Although the `SndDisposeChannel` function always releases memory reserved for sound commands, `SndDisposeChannel` cannot release memory associated with a sound channel record if you have allocated that memory yourself. For example, if you use the `MyCreateSndChannel` function defined in Listing 2-1 to create a sound channel, you must dispose first of the sound channel and then of the memory occupied by the sound channel record, as illustrated in Listing 2-3.

Listing 2-3 Disposing of memory associated with a sound channel

```
FUNCTION MyDisposeSndChannel (sndChan: SndChannelPtr; quietNow: Boolean):
                                    OSErr;
VAR
   myErr:   OSErr;
BEGIN
   myErr := SndDisposeChannel(sndChan, quietNow);  {dispose of channel}
   DisposePtr(Ptr(sndChan));                       {dispose of channel ptr}
   MyDisposeSndChannel := myErr;
END;
```

If you have played a sound resource through a channel, the `SndDisposeChannel` function does not free the memory taken by the resource. You must call the Resource Manager's `ReleaseResource` function to do so, or, if you have detached a resource from a resource file, you could free the memory by making the handle unlocked and purgeable. Note that if you play a sound resource asynchronously, you should not release the memory occupied by the resource until the sound finishes playing or the sound might not play properly. For information on releasing a sound resource after playing a sound asynchronously, see "Playing Sounds Asynchronously" on page 2-46.

IMPORTANT

In Sound Manager versions 3.0 and later, you can play sounds in any number of sound channels. In earlier Sound Manager versions, however, only one kind of sound can be played at one time. This results in several important restrictions on your application. In Sound Manager version 2 and earlier, you should create sound channels just before playing sounds. Once the sound is completed, you should dispose of the channel. If your application is switched out and does not release a sound channel, then other applications may be unable to open sound channels. In particular, the system alert sound might not be heard and the user might not be notified of important system occurrences. In general, while it is acceptable to issue a number of sound commands to the same sound channel, it's not a good idea to play more than one sampled sound on the same sound channel. ▲

Manipulating a Sound That Is Playing

The Sound Manager provides a number of sound commands that you can use to change some of the characteristics of sounds that are currently playing. For example, you can

alter the rate at which a sampled sound is played back, thereby lowering or increasing the pitch of the sound. You can also pause or stop a sound that is currently in progress. See "Pausing and Restarting Sound Channels" on page 2-29 for information on how to pause the processing of a sound channel.

You can use the `getRateCmd` command to determine the rate at which a sampled sound is currently playing. If `SndDoImmediate` returns `noErr` when you pass `getRateCmd`, the current sample rate of the channel is returned as a `Fixed` value in the location that is pointed to by `param2` of the sound command. (As usual, the high bit of that value returned is not interpreted as a sign bit.) Values that specify sampling rates are always interpreted relative to the 22 kHz rate. That is, the `Fixed` value $00010000 indicates a rate of 22 kHz. The value $00020000 indicates a rate of 44 kHz. The value $00008000 indicates a rate of 11 kHz.

To modify the pitch of a sampled sound currently playing, use the `rateCmd` command. The current pitch is set to the rate specified in the `param2` field of the sound command. Listing 2-4 illustrates how to halve the frequency of a sampled sound that is already playing. Note that sending the `rateCmd` command before a sound plays has no effect.

Listing 2-4 Halving the frequency of a sampled sound

```
FUNCTION MyHalveFreq (mySndChan: SndChannelPtr): OSErr;
VAR
   myRate:     LongInt;          {rate of sound play}
   mySndCmd:   SndCommand;       {a sound command}
   myErr:      OSErr;
BEGIN
   {Get the rate of the sample currently playing.}
   mySndCmd.cmd := getRateCmd;   {the command is getRateCmd}
   mySndCmd.param1 := 0;         {unused}
   mySndCmd.param2 := LongInt(@myRate);
   myErr := SndDoImmediate(mySndChan, mySndCmd);

   IF myErr = noErr THEN
   BEGIN
      {Halve the sample rate.}
      mySndCmd.cmd := rateCmd;   {the command is rateCmd}
      mySndCmd.param1 := 0;      {unused}
      mySndCmd.param2 := FixDiv(myRate, $00020000);
      myErr := SndDoImmediate(mySndChan, mySndCmd);
   END;
   MyHalveFreq := myErr;
END;
```

When you halve the frequency of a sampled sound using the technique in Listing 2-4, the sound will play one octave lower than before. In addition, the sound will play twice as

slowly as before. Likewise, if you use the `rateCmd` command to double the frequency of a sound, it plays one octave higher and twice as fast. Using `rateCmd` in this way is like pressing the fast forward button on a tape player while the play button remains depressed.

You can also use `rateCmd` and `getRateCmd` to pause a sampled sound that is currently playing. To do this, read the rate at which it is playing, issue a `rateCmd` command with a rate of 0, and then issue a `rateCmd` command with the previous rate when you want the sound to resume playing.

To change the amplitude (or loudness) of the sound in progress, issue the `ampCmd` command. (See Listing 2-5 for an example.) If no sound is currently playing, `ampCmd` sets the amplitude of the next sound. Specify the desired new amplitude in the `param1` field of the sound command as a value in the range 0 to 255.

Listing 2-5 Changing the amplitude of a sound channel

```
PROCEDURE MySetAmplitude (chan: SndChannelPtr; myAmp: Integer);
VAR
   mySndCmd:   SndCommand;    {a sound command}
   myErr:      OSErr;
BEGIN
   IF chan <> NIL THEN
   BEGIN
      WITH mySndCmd DO
      BEGIN
         cmd := ampCmd;       {the command is ampCmd}
         param1 := myAmp;     {desired amplitude}
         param2 := 0;         {ignored}
      END;
      myErr := SndDoImmediate(chan, mySndCmd);
      IF myErr <> noErr THEN
         DoError(myErr);
   END;
END;
```

If your application has an option that allows users to turn off sound output, you could call the `MySetAmplitude` procedure on all open channels to set the amplitude of all channels to 0. Note that the Sound control panel allows the user to adjust the sound from 0 (softest) to 7 (loudest). This value is independent of the values used for amplitudes of sounds playing in channels, and the Sound Manager uses the Sound control panel value jointly with the amplitude of a sound channel to determine how loudly to play a sound. Sounds with low frequencies sound softer than sounds with high frequencies even if the sounds play at the same amplitude. If the amplitude of a sound is 0, the sound hardware produces no sound; however, when the value set in the Sound control panel is 0, sound might still play, depending on the amplitude.

You can use the `getAmpCmd` command to determine the current amplitude of a sound in progress. The `getAmpCmd` command is similar to `getRateCmd`, except that the value returned is an integer. The value returned in `param2` is in the range 0–255. Listing 2-6 shows an example:

Listing 2-6 Getting the amplitude of a sound in progress

```
VAR
   myAmp:   Integer;
BEGIN
   mySndCmd.cmd := getAmpCmd;
   mySndCmd.param1 := 0;                        {unused}
   mySndCmd.param2 := LongInt(@myAmp);
   myErr := SndDoImmediate(mySndChan, mySndCmd);
END;
```

To modify the timbre of a sound defined using by square-wave data, use the `timbreCmd` command. A sine wave is specified as 0 in `param1` and produces a very clear sound. A value of 254 in `param1` represents a modified square wave and produces a buzzing sound. To avoid a bug in some versions of the Sound Manager, you should not use the value 255. You should change the timbre before playing the sound.

Stopping Sound Channels

The Sound Manager allows you both to stop a sound currently in progress in a channel and to remove all pending sound commands from a channel.

Note
If you have started a sound playing by using the `SndStartFilePlay` function, then you can stop play by using the `SndStopFilePlay` function. See "Managing an Asynchronous Play From Disk" on page 2-52 for more details. ◆

To cause the Sound Manager to stop playing the sound in progress, send the `quietCmd` command. Here's an example:

```
mySndCmd.cmd := quietCmd;               {the command is quietCmd}
mySndCmd.param1 := 0;                   {unused}
mySndCmd.param2 := 0;                   {unused}

{stop the sound now playing}
myErr := SndDoImmediate(mySndChan, mySndCmd, FALSE);
```

To bypass the command queue, you should issue `quietCmd` by using `SndDoImmediate`. Any sound commands that are already in the sound channel remain there, however, and further sound commands can be queued in that channel.

If you wish to flush a sound channel without disturbing any sounds already in progress, issue the `flushCmd` command. Here's an example:

```
mySndCmd.cmd := flushCmd;               {the command is flushCmd}
mySndCmd.param1 := 0;                   {unused}
mySndCmd.param2 := 0;                   {unused}

{flush the channel}
myErr := SndDoImmediate(mySndChan, mySndCmd, FALSE);
```

If you want to stop all sound production by a particular sound channel immediately, you should issue a `flushCmd` command and then a `quietCmd` command. If you issue only a `flushCmd` command, the sound currently playing is not stopped. If you issue only a `quietCmd` command, the Sound Manager stops the current sound but continues with any other queued commands. (By calling `flushCmd` before `quietCmd`, you ensure that no other queued commands are processed.)

Note
The Sound Manager sends a `quietCmd` command when your application calls the `SndDisposeChannel` function. The `quietCmd` command is preceded by a `flushCmd` command if the `quietNow` parameter is `TRUE`. ◆

Pausing and Restarting Sound Channels

If you want to pause command processing in a particular channel, you can use either of two sound commands, `waitCmd` or `pauseCmd`.

Note
If you have started a sound playing by using the `SndStartFilePlay` function, then you can pause and resume play by using the `SndPauseFilePlay` function. See "Managing an Asynchronous Play From Disk" on page 2-52 for more details. ◆

The `waitCmd` command suspends all processing in a channel for a specified number of half-milliseconds. Here's an example:

```
mySndCmd.cmd := waitCmd;                {the command is waitCmd}
mySndCmd.param1 := 2000;                {1-second wait duration}
mySndCmd.param2 := 0;                   {unused}

{pause the channel}
myErr := SndDoImmediate(mySndChan, mySndCmd, FALSE);
```

To pause the processing of commands in a sound channel for an unspecified duration, use the `pauseCmd` command. Unlike `waitCmd`, `pauseCmd` suspends processing for an undetermined amount of time. Processing does not resume until the Sound Manager receives a `resumeCmd` command for the specified channel.

To issue `waitCmd` or `pauseCmd`, you can use either `SndDoImmediate` or `SndDoCommand`, depending on whether you want the suspension of sound channel processing to begin immediately or when the Sound Manager reaches that command in the normal course of reading commands from a sound channel. The `resumeCmd` command, which is simply the opposite of `pauseCmd`, should be issued by using `SndDoImmediate`. Neither `waitCmd` nor `pauseCmd` stops any sound that is currently playing; these commands simply stop further processing of commands queued in the sound channel.

Note
If no other commands are pending in the sound channel after a `resumeCmd` command, the Sound Manager sends an `emptyCmd` command. The `emptyCmd` command is sent only by the Sound Manager and should not be issued by your application. ◆

Synchronizing Sound Channels

You can synchronize several different sound channels by issuing `syncCmd` commands. The `param1` field of the sound command contains a count, and the `param2` field contains an arbitrary identifier. The Sound Manager keeps track of the count for each channel being synchronized. When the Sound Manager receives a `syncCmd` command for a certain channel, it decrements the count for each channel having the given identifier, including the newly synchronized channel. Command processing resumes on a channel when the count becomes 0. Thus, if you know how many channels you need to synchronize, you can synchronize them all by arranging for all of their counts to become zero simultaneously. Listing 2-7 illustrates the use of the `syncCmd` command.

Listing 2-7 Adding a channel to a group of channels to be synchronized

```
PROCEDURE MySync1Chan (chan: SndChannelPtr; count: Integer;
                       identifier: LongInt);
VAR
   mySndCmd:   SndCommand;       {a sound command}
   myErr:      OSErr;
BEGIN
   WITH mySndCmd DO
   BEGIN
      cmd := syncCmd;            {the command is syncCmd}
      param1 := count;
      param2 := identifier;      {ID of group to be synchronized}
   END;
   myErr := SndDoImmediate(chan, mySndCmd);
   IF myErr <> noErr THEN
      DoError(myErr);
END;
```

For example, to synchronize three channels, first create the channels and then call the `MySync1Chan` procedure defined in Listing 2-7 for the first channel with a count equal to 4, for the second channel with a count equal to 3, and for the third channel with a count equal to 2, using the same arbitrary identifier for each call to `MySync1Chan`. Then fill all channels with appropriate sound commands. (For example, you might send commands that will cause the same sequence of notes to be produced on all three synchronized channels.) Finally, call the `MySync1Chan` procedure one final time, passing any of the three channels and a count of 1. By that time, all of the other channels will have counts of 1, and all counts will become 0 simultaneously, thus initiating synchronized play.

Note
The `syncCmd` command is intended to make it easy to synchronize sound channels. You can use the `syncCmd` command to start multiple channels of sampled sound playing simultaneously, but if you require precise synchronization of sampled-sound channels, you might achieve better results with the Time Manager, which is described in *Inside Macintosh: Processes*. ◆

Managing Sound Volumes

Versions of the Sound Manager prior to 3.0 allow you to set only one volume level, which applies to all sounds produced by the audio hardware. The Sound Manager versions 3.0 and later provide greatly improved control over the volumes of the sounds you ask it to create. You can use new facilities to

- set the volumes of the left and right channels of sound independently of each other
- set the volume of the system alert sound
- set the default volume of a particular sound output device

You can set the system alert sound volume to a different level than that of any other sounds you produce. For example, you can set the system alert sound to play at a lower volume than other sounds. This would allow a user to hear QuickTime movies at full volume and to hear system alert sounds at a lower volume.

You can use the `volumeCmd` and `getVolumeCmd` sound commands to set and get the right and left volumes of sound. You specify a channel's volume with 16-bit value, where 0 represents no volume and hexadecimal $0100 represents full volume. The Sound Manager defines constants for silence and full volume.

```
CONST
   kFullVolume                  = $0100;
   kNoVolume                    = 0;
```

The `volumeCmd` sound command expects the right and left volumes to be encoded as the high word and low word, respectively, of `param2`. For example, to set the left channel to half volume and the right channel to full volume, you pass the value $01000080 in `param2`, as illustrated in Listing 2-8.

Listing 2-8 Setting left and right volumes

```
FUNCTION MySetVolume (chan: SndChannelPtr): OSErr;
VAR
   mySndCmd:      SndCommand;
   myRightVol:    Integer;
   myLeftVol:     Integer;
   myErr:         OSErr;
BEGIN
   myRightVol := kFullVolume;
   myLeftVol := kFullVolume DIV 2;
   mySndCmd.cmd := volumeCmd;
   mySndCmd .param1 := 0;              {unused with volumeCmd}
   mySndCmd.param2 := BSL(myRightVol, 16) + myLeftVol;
   myErr := SndDoImmediate(chan, mySndCmd);
   MySetVolume := myErr;
END;
```

You can also use the `volumeCmd` sound command to pan a sound from one side to another. For example, to send the output signal entirely to the right channel, pass the value $01000000 in `param2`. To send the output signal entirely to the left channel, pass the value $00000100 in `param2`. You can overdrive a channel's volume by passing volume levels greater than $0100. For example, to play the left channel of a stereo sound at twice full volume while playing the right channel at full volume, pass the value $01000200.

You can use the `GetSysBeepVolume` and `SetSysBeepVolume` functions to get and set the output volume level of the system alert sound. Any calls to the `SysBeep` procedure use the volume set by the previous call to `SetSysBeepVolume`. As you've learned, this allows you to set a lower volume for the system alert sound than for your other sound output.

You can use the `GetDefaultOutputVolume` and `SetDefaultOutputVolume` functions to set the default output volumes for a particular output device. Each output device has its own current volume setting and its own default setting. If the user changes the output device (using the Sound control panel), the newly selected device will use its own default volume level.

Obtaining Sound-Related Information

Developments in the sound hardware available on Macintosh computers and in the Sound Manager routines that allow you to drive that hardware have made it imperative that your application pay close attention to the sound-related features of the operating environment. For example, some Macintosh computers do not have the sound input hardware necessary to allow sound recording. Similarly, some other Macintosh computers are not able to record sounds and play sounds simultaneously. Before taking

advantage of a sound-related feature that is not available on all Macintosh computers, you should check to make sure that the target machine provides the features you need.

To make appropriate decisions about the sound you want to produce, you might need to know some or all of the following types of information:

- whether a machine can produce stereophonic sounds
- what version of the Sound Manager is available
- whether a machine can play multiple channels of sound, and whether it can take advantage of the enhanced Sound Manager's play-from-disk capabilities
- whether a sound playing from disk is active or paused
- how many channels of sound are currently open
- whether the system beep has been disabled

The following sections describe how to use the `Gestalt` function and Sound Manager routines to determine these types of information.

Obtaining Information About Available Sound Features

You can use the `Gestalt` function to obtain information about a number of hardware- and software-related sound features. For instance, you can use `Gestalt` to determine whether a machine can produce stereophonic sounds and whether it can mix both left and right channels of sound on the internal speaker. Many applications don't need to call `Gestalt` to get this kind of information if they rely on the Sound Manager's ability to produce reasonable sounding output on whatever audio hardware is available. Other applications, however, do need to use `Gestalt` to get this information if they depend on specific hardware or software features that are not available on all Macintosh computers.

To get sound-related information from `Gestalt`, pass it the `gestaltSoundAttr` selector.

```
CONST
   gestaltSoundAttr        = 'snd ';   {sound attributes}
```

If `Gestalt` returns successfully, it passes back to your application a 32-bit value that represents a bit pattern. The following constants define the bits currently set or cleared by `Gestalt`:

```
CONST
   gestaltStereoCapability    = 0;     {built-in hw can play stereo sounds}
   gestaltStereoMixing        = 1;     {built-in hw mixes stereo to mono}
   gestaltSoundIOMgrPresent   = 3;     {sound input routines available}
   gestaltBuiltInSoundInput   = 4;     {built-in input hw available}
   gestaltHasSoundInputDevice = 5;     {sound input device available}
   gestaltPlayAndRecord       = 6;     {built-in hw can play while recording}
   gestalt16BitSoundIO        = 7;     {built-in hw can handle 16-bit data}
   gestaltStereoInput         = 8;     {built-in hw can record stereo sounds}
```

```
gestaltLineLevelInput      = 9;     {built-in input hw needs line level}
gestaltSndPlayDoubleBuffer = 10;    {play from disk routines available}
gestaltMultiChannels       = 11;    {multiple channels of sound supported}
gestalt16BitAudioSupport   = 12;    {16-bit audio data supported}
```

If the bit `gestaltStereoCapability` is `TRUE`, the built-in hardware can play stereo sounds. The bit `gestaltStereoMixing` indicates that the sound hardware of the machine mixes both left and right channels of stereo sound into a single audio signal for the internal speaker. Listing 2-9 demonstrates the use of the `Gestalt` function to determine if a machine can play stereo sounds.

Listing 2-9 Determining if stereo capability is available

```
FUNCTION MyHasStereo: Boolean;
VAR
   myFeature:     LongInt;
   myErr:         OSErr;
BEGIN
   myErr := Gestalt(gestaltSoundAttr, myFeature);
   IF myErr = noErr THEN      {test stereo capability bit}
      MyHasStereo := BTst(myFeature, gestaltStereoCapability)
   ELSE
      MyHasStereo := FALSE;   {no sound features available}
END;
```

As shown in the chapter "Introduction to Sound on the Macintosh," you can determine whether your application can record by testing the `gestaltHasSoundInputDevice` bit. To determine whether a built-in sound input device is available, you can test the `gestaltBuiltInSoundInput` bit. The `gestaltSoundIOMgrPresent` bit indicates whether the sound input routines are available. Because the `gestaltHasSoundInputDevice` bit is not set if the routines are not available, only sound input device drivers should need to use the `gestaltSoundIOMgrPresent` bit.

For a complete description of the response bits set by `Gestalt`, see "Gestalt Selector and Response Bits" beginning on page 2-90.

Obtaining Version Information

The Sound Manager provides functions that allow you to determine the version numbers both of the Sound Manager itself and of the MACE compression and expansion routines. Generally, you should avoid trying to determine which features or routines are present by reading a version number. Usually, the `Gestalt` function (discussed in the previous section) provides a better way to find out if some set of features, such as sound input capability, is available. In some cases, however, you can use these version routines to overcome current limitations of the information returned by `Gestalt`.

Both of these functions return a value of type `NumVersion` that contains the same information as the first 4 bytes of a resource of type `'vers'`. The first and second bytes contain the major and minor version numbers, respectively; the third and fourth bytes contain the release level and the stage of the release level. For most purposes, the major and minor release version numbers are sufficient to identify the version. (See the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials* for a complete discussion of the format of `'vers'` resources.)

You can use the `SndSoundManagerVersion` function to determine which version of the Sound Manager is present. Listing 2-10 shows how to determine if the enhanced Sound Manager is available.

Listing 2-10 Determining if the enhanced Sound Manager is present

```
FUNCTION MyHasEnhancedSoundManager: Boolean;
VAR
   myVersion:     NumVersion;
BEGIN
   IF MyTrapAvailable(_SoundDispatch) THEN
   BEGIN
      myVersion := SndSoundManagerVersion;
      MyHasEnhancedSoundManager := myVersion.majorRev >= 2;
   END
   ELSE
      MyHasEnhancedSoundManager := FALSE
END;
```

The `MyHasEnhancedSoundManager` function defined in Listing 2-10 relies on the `MyTrapAvailable` function, which is an application-defined routine provided in *Inside Macintosh: Operating System Utilities*. If the `_SoundDispatch` trap is not available, the `SndSoundManagerVersion` function is not available either, in which case the enhanced Sound Manager is certainly not available.

You can use the `MACEVersion` function to determine the version number of the available MACE routines (for example, `Comp3to1`).

Testing for Multichannel Sound and Play-From-Disk Capabilities

The ability to play multiple channels of sound simultaneously and the ability to initiate plays from disk were first introduced with the enhanced Sound Manager. Even with the enhanced Sound Manager, however, these capabilities are present only on computers equipped with suitable sound output hardware (such as an Apple Sound Chip). Sound Manager version 3.0 defines 2 additional bits in the `Gestalt` response parameter that allow you to test directly for these two capabilities.

```
CONST
   gestaltSndPlayDoubleBuffer    = 10; {play from disk routines available}
   gestaltMultiChannels          = 11; {multiple channels of sound supported}
```

Ideally, it should be sufficient to test directly, using `Gestalt`, for either multichannel sound capability or play-from-disk capability. If your application happens to be running under the enhanced Sound Manager, however, the two new response bits are not defined. In that case, you'll need to test also whether the Apple Sound Chip is available, because multichannel sound and play from disk are supported by the enhanced Sound Manager only if the Apple Sound Chip is available. To test for the presence of the Apple Sound Chip, you can use the `Gestalt` function with the `gestaltHardwareAttr` selector and the `gestaltHasASC` bit. Listing 2-11 combines these two tests into a single routine that returns `TRUE` if the computer supports multichannel sound.

Listing 2-11 Testing for multichannel play capability

```
FUNCTION MyCanPlayMultiChannels: Boolean;
VAR
   myResponse:    LongInt;
   myResult:      Boolean;
   myErr:         OSErr;
   myVersion:     NumVersion;
BEGIN
   myResult := FALSE;
   myVersion := SndSoundManagerVersion;
   myErr := Gestalt(gestaltSoundAttr, myResponse);
   IF myVersion.majorRev >= 3 THEN
      IF (myErr = noErr) AND (BTst(myResponse, gestaltMultiChannels)) THEN
         myResult := TRUE
   ELSE
      BEGIN
         myErr := Gestalt(gestaltHardwareAttr, myResponse);
         IF (myErr = noErr) AND (BTst(myResponse, gestaltHasASC)) THEN
            myResult := TRUE
      END;
   MyCanPlayMultiChannels := myResult;
END;
```

The function `MyCanPlayMultiChannels` first tries to get the desired information by calling the `Gestalt` function with the `gestaltSoundAttr` selector. If `Gestalt` returns successfully and the `gestaltMultiChannels` bit is set in the `response` parameter, then multichannel play capability is present. Notice that the multichannel bit is checked only if the version of the Sound Manager is 3.0 or greater. If the version is not at least 3.0, then `MyCanPlayMultiChannels` calls the `Gestalt` function with the

gestaltHardwareAttr selector. If the computer contains the Apple Sound Chip, then again multichannel play capability is present.

Note
The gestaltHasASC bit is set only on machines that contain an Apple Sound Chip. You should test for the presence of the Apple Sound Chip only in the circumstances described above. ◆

You could write a similar function to test for the ability to initiate a play from disk. Listing 2-12 shows an example.

Listing 2-12 Testing for play-from-disk capability

```
FUNCTION HasPlayFromDisk: Boolean;
VAR
   myResponse:    LongInt;
   myResult:      Boolean;
   myErr:         OSErr;
   myVersion:     NumVersion;
BEGIN
   myResult := FALSE;
   myVersion := SndSoundManagerVersion;
   myErr := Gestalt(gestaltSoundAttr, myResponse);
   IF myVersion.majorRev >= 3 THEN
      IF (myErr = noErr) AND
                        (BTst(myResponse, gestaltSndPlayDoubleBuffer)) THEN
         myResult := TRUE
   ELSE
      BEGIN
         myErr := Gestalt(gestaltHardwareAttr, myResponse);
         IF (myErr = noErr) AND (BTst(myResponse, gestaltHasASC)) THEN
            myResult := TRUE
      END;
   HasPlayFromDisk := myResult;
END;
```

Obtaining Information About a Single Sound Channel

You can use the SndChannelStatus function to obtain information about a single sound channel and about the status of a disk-based playback on that channel, if one exists. For example, you can use SndChannelStatus to determine if a channel is being used for play from disk, how many seconds of the sound have been played, and how many seconds remain to be played.

One of the parameters required by the `SndChannelStatus` function is a pointer to a sound channel status record, which you must allocate before calling `SndChannelStatus`. A sound channel status record has this structure:

```
TYPE SCStatus =
RECORD
   scStartTime:            Fixed;      {starting time for play from disk}
   scEndTime:              Fixed;      {ending time for play from disk}
   scCurrentTime:          Fixed;      {current time for play from disk}
   scChannelBusy:          Boolean;    {TRUE if channel is processing cmds}
   scChannelDisposed:      Boolean;    {reserved}
   scChannelPaused:        Boolean;    {TRUE if channel is paused}
   scUnused:               Boolean;    {unused}
   scChannelAttributes:    LongInt;    {attributes of this channel}
   scCPULoad:              LongInt;    {CPU load for this channel}
END;
```

The `scStartTime`, `scEndTime`, and `scCurrentTime` fields are 0 unless the Sound Manager is currently playing from disk through the specified channel. If a play from disk is occurring, the `scStartTime` and `scEndTime` fields reflect the starting and ending points of the play, defined in seconds; the `scCurrentTime` field indicates the number of seconds between the beginning of the sound on disk and the part of the sound currently being played. The Sound Manager sets the values of the `scStartTime` and `scEndTime` fields based on the values you set in an audio selection record. (See page 2-100 for a description of the audio selection record.)

Note that because the Sound Manager might be playing only a selection of a sound, the `scCurrentTime` field does not reflect the number of seconds of sound play that have elapsed. To compute the number of seconds of sound play elapsed, you can subtract the value in the `scStartTime` field from that in the `scCurrentTime` field. However, because the Sound Manager updates the value of the `scCurrentTime` field only periodically, you should not rely on the accuracy of its value.

The `scChannelBusy` and `scChannelPaused` fields reflect whether a channel is processing commands and whether a channel is paused, respectively. After issuing a series of sound commands, you can use these fields to determine if the channel has finished processing all of the commands. If both `scChannelBusy` and `scChannelPaused` are `FALSE`, the Sound Manager has processed all of the channel's commands.

You can mask out certain values in the `scChannelAttributes` field to determine how a channel has been initialized.

```
CONST
   initPanMask    = $0003;    {mask for right/left pan values}
   initSRateMask  = $0030;    {mask for sample rate values}
   initStereoMask = $00C0;    {mask for mono/stereo values}
```

The `scCPULoad` field previously reflected the percentage of CPU processing power used by the sound channel. However, this field is obsolete, and you should not rely on its value.

Listing 2-13 illustrates the use of the `SndChannelStatus` function. It defines a function that takes a sound channel pointer as a parameter and determines whether a disk-based playback on that channel is paused.

Listing 2-13 Determining whether a sound channel is paused

```
FUNCTION MyChannelIsPaused (chan: SndChannelPtr): Boolean;
VAR
   myErr:         OSErr;
   mySCStatus:    SCStatus;
BEGIN
   MyChannelIsPaused := FALSE;
   myErr := SndChannelStatus(chan, Sizeof(SCStatus), @mySCStatus);
   IF myErr = noErr THEN
      MyChannelIsPaused := mySCStatus.scChannelPaused;
END;
```

The function defined in Listing 2-13 simply reads the `scChannelPaused` field to see if the playback is currently paused.

Note
In Sound Manager versions earlier than 3.0, pausing a sound channel by issuing a `pauseCmd` command does not change the `scChannelPaused` field. The `scChannelPaused` field is `TRUE` only if the Sound Manager is executing a disk-based playback on the channel and that playback is paused by the `SndPauseFilePlay` function. This problem is fixed in Sound Manager versions 3.0 and later. ◆

Obtaining Information About All Sound Channels

You can use the `SndManagerStatus` function to determine information about all the sound channels that are currently allocated by all applications. For example, you can use this function to determine how many channels are currently allocated. One of the parameters required by the `SndManagerStatus` function is a pointer to a Sound Manager status record, which you must allocate before calling `SndManagerStatus`. A Sound Manager status record has this structure:

```
TYPE SMStatus =
PACKED RECORD
   smMaxCPULoad:     Integer;    {maximum load on all channels}
   smNumChannels:    Integer;    {number of allocated channels}
   smCurCPULoad:     Integer;    {current load on all channels}
END;
```

The `smNumChannels` field contains the number of sound channels currently allocated. This does not mean that the channels are actually being used, only that they have been created with the `SndNewChannel` function and not yet disposed.

The Sound Manager uses information that it returns in the `smMaxCPULoad` and `smCurCPULoad` fields to help it determine whether it can allocate a new channel when your application calls the `SndNewChannel` function. The Sound Manager sets `smMaxCPULoad` to a default value of 100 at startup time, and the `smCurCPULoad` field reflects the approximate percentage of CPU processing power currently taken by allocated sound channels.

▲ **WARNING**
Your application should not reply on the values returned in the `smMaxCPULoad` and `smCurCPULoad` fields. To determine if it is safe to allocate a channel, simply try to allocate it with the `SndNewChannel` function. That function returns the appropriate result code if allocating the channel would put too much of a strain on CPU processing. ▲

Listing 2-14 illustrates the use of `SndManagerStatus`. It defines a function that returns the number of sound channels currently allocated by all applications.

Listing 2-14 Determining the number of allocated sound channels

```
FUNCTION MyGetNumChannels: Integer;
VAR
   myErr:         OSErr;
   mySMStatus:    SMStatus;
BEGIN
   MyGetNumChannels := 0;
   myErr := SndManagerStatus (Sizeof(SMStatus), @mySMStatus);
   IF myErr = noErr THEN
      MyGetNumChannels := mySMStatus.smNumChannels;
END;
```

Determining and Changing the Status of the System Alert Sound

The enhanced Sound Manager includes two routines—`SndGetSysBeepState` and `SndSetSysBeepState`—that allow you to determine and alter the status of the system alert sound. You might wish to disable the system alert sound if you are playing sound and need to ensure that the sound you are playing is not interrupted. Currently, two states are defined:

```
CONST
   sysBeepDisable       = $0000;    {system alert sound disabled}
   sysBeepEnable        = $0001;    {system alert sound enabled}
```

You can determine the status of the system alert sound like this:

```
SndGetSysBeepState(currentState);
```

And you can disable the system alert sound like this:

```
myErr := SndSetSysBeepState(sysBeepDisable);
```

When the system alert sound is disabled, the Sound Manager effectively ignores all calls to the `SysBeep` procedure. No sound is created and the menu bar does not flash. Also, no resources are loaded into memory.

Note
Even when the system alert sound is enabled, it's possible that the system alert sound will not play; for example, the speaker volume might be set to 0, or playing the requested system alert sound might require too much CPU time. In such a case, the menu bar flashes. ◆

By default, the system alert sound is enabled. If you disable the system alert sound so that your application can play a sound without being interrupted, be sure to enable the sound when your application receives a suspend event or when the user quits your application.

Playing Notes

You can play notes one at a time by using the `SndDoCommand` or `SndDoImmediate` function to issue `freqDurationCmd` sound commands. A sound plays for a specified duration at a specified frequency. You can play sounds defined by any of the three sound data formats. If you play wave-table data or sampled-sound data, then a voice must previously have been installed in the channel. (See "Installing Voices Into Channels" on page 2-43 for instructions on installing wave tables and sampled sounds as voices.)

You can also play notes by issuing the `freqCmd` command, which is identical to the `freqDurationCmd` command, except that no duration is specified when you issue `freqCmd`.

Note
A `freqDurationCmd` command might in certain cases continue playing until another command is available in the sound channel. Therefore, to play a single note for a specified duration, you should issue `freqDurationCmd` followed immediately by `quietCmd`. See "Stopping Sound Channels" on page 2-28 for further details on `quietCmd`. ◆

The structure of a `freqDurationCmd` command is slightly different from that of most other sound commands. The `param1` field contains the duration of the sound, specified in half-milliseconds. A value of 2000 represents a duration of 1 second. The maximum duration is 32,767, or about 16 seconds, in Sound Manager versions 2.0 and earlier; the maximum duration in Sound Manager version 3.0 and later is 65,536, or about 32 seconds. The `param2` field specifies the frequency of the sound. The frequency is specified as a MIDI note value (that is, a value defined by the established MIDI

standard). Listing 2-15 uses the `freqDurationCmd` command in a way that ensures the sound stops after the specified duration.

Listing 2-15 Using the `freqDurationCmd` command

```
PROCEDURE MyPlayFrequencyOnce (mySndChan: SndChannelPtr;
                               myMIDIValue: Integer;
                               milliseconds: Integer);
CONST
   kNoWait = TRUE;                  {add now to full queue?}
VAR
   mySndCmd:   SndCommand;          {a sound command}
   myErr:      OSErr;
BEGIN
   {Start the sound playing.}
   WITH mySndCmd DO
   BEGIN
      cmd := freqDurationCmd;       {play for period of time}
      param1 := milliseconds * 2;   {half-milliseconds}
      param2 := myMIDIValue;        {MIDI value to play}
   END;
   myErr := SndDoCommand(mySndChan, mySndCmd, NOT kNoWait);
   IF myErr <> noErr THEN
      DoError(myErr)
   ELSE
   BEGIN                            {ensure that sound stops}
      WITH mySndCmd DO
      BEGIN
         cmd := quietCmd;           {stop playing sound}
         param1 := 0;               {unused with quietCmd}
         param2 := 0;               {unused with quietCmd}
      END;
      myErr := SndDoCommand(mySndChan, mySndCmd, NOT kNoWait);
      IF myErr <> noErr THEN
         DoError(myErr);
   END;
END;
```

Table 2-2 shows the decimal values that can be sent with a `freqDurationCmd` or `freqCmd` command. Middle C is represented by a value of 60 and is defined by a special Sound Manager constant.

```
CONST
   kMiddleC       = 60;       {MIDI note value for middle C}
```

Other specifiable frequencies correspond to MIDI note values.

Table 2-2 Frequencies expressed as MIDI note values

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1				0	1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	60	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

You can play square-wave and wave-table data at these frequencies only. If you are playing a sampled sound, however, you can modify the `sampleRate` field of the sound header to play a sound at an arbitrary frequency. To do so, use the following formula:

new sample rate = (new frequency / original frequency) * original sample rate

where the new and original frequencies are measured in hertz. To convert a MIDI value to hertz for use in this formula, note that middle C is defined as 261.625 Hz and that the ratio between the frequencies of consecutive MIDI values equals the twelfth root of 2, defined by the constant `twelfthRootTwo`.

```
CONST
   twelfthRootTwo          = 1.05946309434;
```

IMPORTANT

When calculating with numbers of type `Fixed`, pay attention to possible overflows. The maximum value of a number of type `Fixed` is 65,535.0. As a result, some sample rates and pitches cannot be specified. Sound Manager version 3.0 fixes these overflow problems. ▲

You can rest a channel for a specified duration by issuing a `restCmd` command. The duration, specified in half-milliseconds, is passed in the `param1` field of the sound command.

Installing Voices Into Channels

You can play frequencies defined by any of the three sound data types. By playing a frequency defined by wave-table or sampled-sound data, you can achieve a different

sound than by playing that same frequency using square-wave data. For example, you might wish to play the sound of a dog's barking at a variety of frequencies. To do that, however, you need to install a voice of the barking into the sound channel to which you want to send `freqCmd` or `freqDurationCmd` commands.

You can install a wave table into a channel as a voice by issuing the `waveTableCmd` command. The `param1` field of the sound command specifies the length of the wave table, and the `param2` field is a pointer to the wave-table data itself. Note that the Sound Manager resamples the wave table so that it is exactly 512 bytes long.

You can install a sampled sound into a channel as a voice by issuing the `soundCmd` command. You can either issue this command from your application or put it into an `'snd '` resource. If your application sends this command, `param2` is a pointer to the sampled sound locked in memory. If `soundCmd` is contained within an `'snd '` resource, the high bit of the command must be set. To use a sampled-sound `'snd '` as a voice, first obtain a pointer to the sampled sound header locked in memory. Then pass this pointer in `param2` of a `soundCmd` command. After using the sound, your application is expected to unlock this resource and allow it to be purged.

Listing 2-16 demonstrates how you can use the `soundCmd` command to install a sampled sound in memory as a voice in a channel.

Listing 2-16 Installing a sampled sound as a voice in a channel

```
FUNCTION MyInstallSampledVoice (mySndHandle: Handle;
                                mySndChan: SndChannelPtr): OSErr;
VAR
   mySndCmd:         SndCommand;          {a sound command}
   mySndHeader:      SoundHeaderPtr;      {sound header from resource}
BEGIN
                                          {get pointer to sound header}
   mySndHeader := MyGetSoundHeader(mySndHandle);
   WITH mySndCmd DO
   BEGIN
      cmd := soundCmd;                    {install sampled voice}
      param1 := 0;                        {ignored with soundCmd}
      param2 := LongInt(mySndHeader);     {store sound header location}
   END;
   IF mySndHeader = NIL THEN              {check for defective handle}
      MyInstallSampledVoice := badFormat
   ELSE                                   {install sound as voice}
      MyInstallSampledVoice := SndDoImmediate(mySndChan, mySndCmd);
END;
```

Listing 2-16 relies on the `MyGetSoundHeader` function to obtain a pointer to the sound header within the sound handle. That function is defined in "Obtaining a Pointer to a

Sound Header" on page 2-57 and returns `NIL` if the sound handle does not include a sound header. Note that the `MyGetSoundHeader` function locks the sound handle in memory so that the pointer to the sound header remains valid. When you are done with the sound channel in which you have installed the sampled sound, you should unlock the sound handle and make it purgeable so that it does not waste memory.

Looping a Sound Indefinitely

If you install a sampled sound as a voice in a channel and then play the sound using a `freqCmd` or `freqDurationCmd` command that lasts longer than the sound, the sound will ordinarily stop before the end of the time specified by the `freqCmd` or `freqDurationCmd` command. Sometimes, however, this might not be what you'd like to have happen. For example, you might have recorded the sound of a violin playing and then stored that sound in a resource so that you could play the sound of a violin at a number of different frequencies. Although you could record the sound so that it is long enough to continue playing through the longest `freqCmd` or `freqDurationCmd` command that your application might require, this might not be practical. Fortunately, the Sound Manager provides a mechanism that allows you to repeat sections of sampled sound after the sound has finished playing once completely.

When you use the `freqDurationCmd` command with a sampled sound as the voice, `freqDurationCmd` starts at the beginning of the sampled sound. If necessary to achieve the desired duration of sound, the command replays that part of the sound that is between the loop points specified in the sampled sound header. Note that any sound preceding or following the loop points will not be replayed. There must be an ending point for the loop specified in the header in order for `freqDurationCmd` to work properly.

Listing 2-17 Looping an entire sampled sound

```
PROCEDURE MyDoLoopEntireSound (sndHandle: Handle);
VAR
   mySndHeader:   SoundHeaderPtr;      {sound header from resource}
   myTotalBytes:  LongInt;             {bytes of data to loop}
BEGIN
   mySndHeader := MyGetSoundHeader(sndHandle);
   IF mySndHeader <> NIL THEN
   BEGIN                               {compute bytes of sound data}
      CASE mySndHeader^.encode OF
         stdSH:                        {standard sound header}
            WITH mySndHeader^ DO
               myTotalBytes := mySndHeader^.length;
         extSH:                        {extended sound header}
            WITH ExtSoundHeaderPtr(mySndHeader)^ DO
               myTotalBytes := numChannels * numFrames * (sampleSize DIV 8);
         cmpSH:                        {compressed sound header}
```

```
            WITH CmpSoundHeaderPtr(mySndHeader)^ DO
               myTotalBytes := numChannels * numFrames * (sampleSize DIV 8);
      END;
      WITH mySndHeader^ DO
      BEGIN                           {set loop points}
         loopStart := 0;              {start with first byte}
         loopEnd := myTotalBytes - 1;  {end with last byte}
      END;
   END;
END;
```

Listing 2-17 uses the `MyGetSoundHeader` function defined in "Obtaining a Pointer to a Sound Header" on page 2-57. Note that the formula for computing the length of a sound depends on the type of sound header. Also, while the formula is the same for both an extended and a compressed sound header, you must write code that differentiates between the two types of sound headers because the `sampleSize` field is not stored in the same location in both sound headers.

Playing Sounds Asynchronously

The Sound Manager currently allows you to play sounds asynchronously only if you allocate sound channels yourself, using techniques described in "Managing Sound Channels" on page 2-19. But if you use such a technique, your application will need to dispose of a sound channel whenever the application finishes playing a sound. In addition, your application might need to release a sound resource that you played on a sound channel.

To avoid the problem of not knowing when to dispose of a sound channel playing a sound asynchronously, your application could simply allocate a single sound channel when it starts up (or receives a resume event) and dispose of the channel when the user quits (or the application receives a suspend event). However, this solution will not work if you need to release a resource when a sound finishes playing. Also, you might not want to keep a sound channel allocated when you are not using it. For instance, you might want to use the memory taken up by a sound channel for other tasks when no sound is playing.

Your application could call the `SndChannelStatus` function once each time through its main event loop to determine if a channel is still making sound. When the `scBusy` field of the sound channel status record becomes `FALSE`, your application could then dispose of the channel. This technique is easy, but calling `SndChannelStatus` frequently uses up processing time unnecessarily.

The Sound Manager provides other mechanisms that allow your application to find out when a sound finishes playing, so that your application can arrange to dispose of sound channels no longer being used and of other data (such as a sound resource) that you no longer need after disposing of a channel. If you are using the `SndPlay` function or low-level commands to play sound in a channel, then you can use callback procedures. If you are using the `SndStartFilePlay` function to play sound in a channel, then you

can use completion routines. The following sections illustrate how to use callback procedures and completion routines.

Note
Callback procedures are a form of completion routine. However, for clarity, this section uses the terminology "completion routine" only for the routines associated with the `SndStartFilePlay` function. ◆

Using Callback Procedures

This section shows how you can use callback procedures to play one sound asynchronously at a given time. "Managing Multiple Sound Channels" on page 2-53 expands the techniques in this section to show how you can play several asynchronous sounds simultaneously.

The `SndNewChannel` function allows you to associate a callback procedure with a sound channel. For example, the following code opens a new sound channel for which memory has already been allocated and associates it with the callback procedure `MyCallBack`:

```
myErr := SndNewChannel(gSndChan, sampledSynth, initMono, @MyCallback);
```

After filling a channel created by `SndNewChannel` with various commands to create sound, you can then issue a `callBackCmd` command to the channel. When the Sound Manager encounters a `callBackCmd` command, it executes your callback procedure. Thus, by placing the `callBackCmd` command last in a channel, you can ensure that the Sound Manager executes your callback procedure only after it has processed all of the channel's other sound commands.

Note
Be sure to issue `callBackCmd` commands with the `SndDoCommand` function and not the `SndDoImmediate` function. If you issue a `callBackCmd` command with `SndDoImmediate`, your callback procedure might be called before other sound commands you have issued finish executing. ◆

A callback procedure has the following syntax:

```
PROCEDURE MyCallBack (chan: SndChannelPtr; cmd: SndCommand);
```

Because the callback procedure executes at interrupt time, it cannot access its application global variables unless the application's A5 world is set correctly. (For more information on the A5 world, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory*.) When called, the callback procedure is passed two parameters: a pointer to the sound channel that received the `callBackCmd` command and the sound command that caused the callback procedure to be called. Applications can use `param1` or `param2` of the sound command as flags to pass information or instructions to the callback procedure. If your callback procedure is to use your application's global data storage, it must first reset A5 to your application's A5 and then restore it on exit. For example, Listing 2-18 illustrates how to set up a `callBackCmd` command that contains the

required A5 information in the `param2` field. The `MyInstallCallback` function defined there must be called at a time when your application's A5 world is known to be valid.

Listing 2-18 Issuing a callback command

```
FUNCTION MyInstallCallback (mySndChan: SndChannelPtr): OSErr;
CONST
   kWaitIfFull = TRUE;           {wait for room in queue}
VAR
   mySndCmd:   SndCommand;       {a sound command}
BEGIN
   WITH mySndCmd DO
   BEGIN
      cmd := callBackCmd;        {install the callback command}
      param1 := kSoundComplete;  {last command for this channel}
      param2 := SetCurrentA5;    {pass the callback the A5}
   END;
   MyInstallCallback := SndDoCommand(mySndChan, mySndCmd, kWaitIfFull);
END;
```

In this function, `kSoundComplete` is an application-defined constant that indicates that the requested sound has finished playing. You could define it like this:

```
CONST
   kSoundComplete          = 1;     {sound is done playing}
```

Because `param2` of a sound command is a long integer, Listing 2-18 uses it to pass the application's A5 to the callback procedure. That allows the callback procedure to gain access to the application's A5 world.

Note
You can also pass information to a callback routine in the `userInfo` field of the sound channel. ◆

The sample callback procedure defined in Listing 2-19 can thus set A5 to access the application's global variables.

Listing 2-19 Defining a callback procedure

```
PROCEDURE MyCallback (theChan: SndChannelPtr; theCmd: SndCommand);
VAR
   myA5:          LongInt;
BEGIN
   IF theCmd.param1 = kSoundComplete THEN
```

```
   BEGIN
      myA5 := SetA5(theCmd.param2);    {set my A5}
      gCallbackPerformed := TRUE;      {set a global flag}
      myA5 := SetA5(myA5);             {restore the original A5}
   END;
END;
```

▲ **WARNING**
Callback procedures are called at interrupt time and therefore must not attempt to allocate, move, or dispose of memory, dereference an unlocked handle, or call other routines that do so. Also, assembly-language programmers should note that a callback procedure is a Pascal procedure and must preserve all registers other than A0–A1 and D0–D2. ▲

Callback procedures cannot dispose of channels themselves, because that involves disposing of memory. To circumvent this restriction, the callback procedure in Listing 2-19 simply sets the value of a global flag variable that your application defines. Then, once each time through its main event loop, your application must call a routine that checks to see if the flag is set. If the flag is set, the routine should dispose of the channel, release any other memory allocated specifically for use in the channel, and reset the flag variable. Listing 2-20 defines such a routine. Your application should call it once each time through its main event loop.

Listing 2-20 Checking whether a callback procedure has executed

```
PROCEDURE MyCheckSndChan;
CONST
   kQuietNow = TRUE;               {need to quiet channel?}
VAR
   myErr:   OSErr;
BEGIN
   IF gCallbackPerformed THEN      {check global flag}
   BEGIN                           {channel is done}
      gCallbackPerformed := FALSE; {reset global flag}
      IF gSndChan^.userInfo <> 0 THEN
      BEGIN                        {release sound data}
         HUnlock(Handle(gSndChan^.userInfo));
         HPurge(Handle(gSndChan^.userInfo));
      END;
      myErr := MyDisposeSndChannel(gSndChan, kQuietNow);
      gSndChan := NIL;             {set pointer to NIL}
   END;
END;
```

The MyCheckSndChan procedure defined in Listing 2-20 checks the userInfo field of the sound channel to see if it contains the address of a handle. Thus, if you would like the MyCheckSndChan procedure to release memory associated with a sound handle, you need only put the address of the handle in the userInfo field of the sound channel. (If you do not want the MyCheckSndChan procedure to release memory associated with a handle, then you should set the userInfo field to 0 when you allocate the channel. The MyCreateSndChannel function defined in Listing 2-1 on page 2-20 automatically sets this field to 0.) After releasing the memory associated with the sound handle, the MyCheckSndChan procedure calls the MyDisposeSndChannel function (defined in Listing 2-3 on page 2-25) to release the memory occupied by both the sound channel and the sound channel record.

To ensure that the MyCheckSndChan procedure defined in Listing 2-20 does not attempt to dispose a channel before you have created one, you should initialize the gCallbackPerformed variable to FALSE. Also, you should initialize the gSndChan variable to NIL, so that other parts of your application can check to see if a sound is playing simply by checking this variable. For example, if your application must play a sound but another sound is currently playing, you might ensure that the application gives priority to the newer sound by stopping the old one. Listing 2-21 defines a procedure that stops the sound that is playing.

Listing 2-21 Stopping a sound that is playing asynchronously

```
PROCEDURE MyStopPlaying;
BEGIN
   IF gSndChan <> NIL THEN          {is sound really playing?}
      gCallbackPerformed := TRUE;   {set global flag}
   MyCheckSndChan;                  {call routine to do disposing}
END;
```

Once you have defined a callback procedure, a routine that installs the callback procedure, a routine that checks the status of the callback procedure, and a routine that can stop sound play, you need only allocate a sound channel, call the SndPlay function, and install your callback procedure to start an asynchronous sound play. Listing 2-22 defines a procedure that starts an asynchronous play.

Listing 2-22 Starting an asynchronous sound play

```
PROCEDURE MyStartPlaying (mySndID: Integer);
CONST
   kAsync = TRUE;             {play is asynchronous}
VAR
   mySndHandle:   Handle;     {handle to an 'snd ' resource}
   myErr:         OSErr;
BEGIN
```

```
   IF gSndChan <> NIL THEN                {check if channel is active}
      MyStopPlaying;
   gSndChan := MyCreateSndChannel(0, 0, @MyCallbackProc, stdQLength);
   mySndHandle := GetResource('snd ', mySndID);
   IF (mySndHandle <> NIL) AND (gSndChan <> NIL) THEN
   BEGIN                        {start sound playing}
      DetachResource(mySndHandle);        {detach resource from file}
                                          {remember to release sound handle}
      gSndChan^.userInfo := LongInt(mySndHandle);
      HLock(mySndHandle);                 {lock the resource data}
      myErr := SndPlay(gSndChan, mySndHandle, kAsync);
      IF myErr = noErr THEN
         myErr := MyInstallCallback(gSndChan);
      IF myErr <> noErr THEN
         DoError(myErr);
   END;
END;
```

The `MyStartPlaying` procedure uses the `MyCreateSndChannel` function defined in Listing 2-1 to create a sound channel, requesting that the function allocate a standard-sized sound channel command queue. By using such a queue, you can be sure that your application can play any sound resource that contains up to 127 sound commands. If you are sure that your application will play only sampled-sound resources created by the Sound Input Manager, you should request a queue of only two sound commands, thereby leaving enough room for just the `bufferCmd` command contained within the sound resource and the `callBackCmd` command that your application issues.

Before playing the sound, the `MyStartPlaying` procedure defined in Listing 2-22 detaches the sound resource from its resource file after loading it. This is important if the resource file could close while the sound is still playing, or if your application might create another sound channel to play the same sound resource while the sound is still playing.

Synchronizing Sound With Other Actions

If your application uses callback procedures to play sound asynchronously, you might wish to synchronize sound play with other activity, such as an onscreen animation.

Callback procedures allow your application to do that by using different constant values in the `param1` field of the callback command. For example, you could define a constant `kFirstSoundFinished` to signal to your application that the first of a series of sounds has finished playing. Then, your callback procedure could set an appropriate global flag depending on whether the `param1` field equals `kFirstSoundFinished`, `kSoundComplete`, or some other constant that your application defines. Finally, a procedure that you call once each time through your application's event loop could check to see which of the various global flag variables are set and respond appropriately. Meanwhile, sound continues to play.

Managing an Asynchronous Play From Disk

The Sound Manager allows you to play a sound file asynchronously with the `SndStartFilePlay` function by defining a completion routine that sets a global flag to alert the application to dispose of the sound channel when the sound is done playing. Completion routines are thus similar to callback procedures, but they are easier to use in that you do not need to install them. The Sound Manager automatically executes them when a play from disk ends, whether it has ended because the application called the `SndStopFilePlay` function, because the application disposed of the sound channel in which the sound was playing, or because the sound has finished playing.

You define a completion routine like this:

```
PROCEDURE MySoundCompletionRoutine (chan: SndChannelPtr);
```

Note that unlike callback procedures, completion routines have only one parameter, a pointer to a sound channel. Thus, for the completion routine to set the application's A5 world properly, you should pass the value of the application's A5 in the `userInfo` field of the sound channel, like this:

```
gSndChan^.userInfo := SetCurrentA5;
```

Then your completion routine can look in the `userInfo` field of the sound channel to set A5 correctly before it can access any application global variables. Listing 2-23 defines a completion routine that sets A5 correctly.

Listing 2-23 Defining a completion routine

```
PROCEDURE MySoundCompletionRoutine (chan: SndChannelPtr);
VAR
   myA5:    LongInt;
BEGIN
   myA5 := SetA5(chan^.userInfo);      {set my A5}
   gCompletionPerformed := TRUE;       {set a global flag}
   myA5 := SetA5(myA5);                {restore the original A5}
END;
```

The completion routine defined in Listing 2-23 sets a global flag variable to indicate that the completion routine has been called. To start a sound file playing, you can use a routine analogous to that defined in Listing 2-22, but when allocating a sound channel, you need only allocate a queue of a single sound command. You can than use a procedure analogous to that defined in Listing 2-20 to check the flag once each time through the application's event loop and dispose of the sound channel if the flag is set.

If you do use the `SndStartFilePlay` function to play sounds asynchronously, then you can pause, restart, and stop play simply by using the `SndPauseFilePlay` and `SndStopFilePlay` functions.

You use SndPauseFilePlay to temporarily suspend a sound from playing. If a sound is playing and you call SndPauseFilePlay, then the sound is paused. If the sound is paused and you call SndPauseFilePlay again, then the sound resumes playing. Hence, the SndPauseFilePlay routine acts like a pause button on a tape player, which toggles the tape between playing and pausing. (You can determine the current state of a play from disk by using the SndChannelStatus function. See "Obtaining Information About a Single Sound Channel" on page 2-37 for more details.) Finally, you can use SndStopFilePlay to stop the file from playing.

Playing Selections

The sixth parameter passed to the SndStartFilePlay function is a pointer to an **audio selection record,** which allows you to specify that only part of the sound be played. If that parameter has a value different from NIL, then SndStartFilePlay plays only a specified selection of the entire sound. You indicate which part of the entire sound to play by giving two offsets from the beginning of the sound, a time at which to start the selection and a time at which to end the selection. Currently, both time offsets must be specified in seconds.

Here is the structure of an audio selection record:

```
TYPE AudioSelection =
PACKED RECORD
   unitType:   LongInt;    {type of time unit}
   selStart:   Fixed;      {starting point of selection}
   selEnd:     Fixed;      {ending point of selection}
END;
```

To play a selection, you should specify in the selStart and selEnd fields the starting and ending point in seconds of the sound to play. Also, you must set the unitType field to the constant unitTypeSeconds.

If you wish to play an entire sound, you can simply pass NIL to the SndStartFilePlay function. Alternatively, you can set the unitType field to the constant unitTypeNoSelection, in which case the values in the selStart and selEnd fields are ignored.

Managing Multiple Sound Channels

If you are writing an application that can play multiple channels of sound on Macintosh computers that support that feature, you can use the Sound Manager's asynchronous playing abilities, but you might encounter some special obstacles. The technique for playing sounds asynchronously described in "Playing Sounds Asynchronously" on page 2-46 has a limitation if you are using multiple sound channels. Using that technique without modification, you would need to define each separate sound channel in a different global variable, and you would need to use several global flags in your callback procedure to signal which sound channels have finished processing sound commands.

Although it is easy to modify the code in "Playing Sounds Asynchronously" to use several flags, this solution might not be satisfactory for an application in which the number of sound channels open can vary. For example, suppose that you are writing entertainment software with dozens of sound effects that correspond to actions on the screen and you wish to use the Sound Manager asynchronously so that several sound effects can be played at once. It would be cumbersome to associate a separate global sound channel variable with each sound and create a flag variable for each of these sound channels. Also, you might wish to play the same sound simultaneously in two separate channels. It would be better to write code that manages a global list of sound channels and then provides a simple routine that allows you to add a channel to the list. This section shows how you might implement such a list of sound channels. Listing 2-24 defines a data structure that you could use to track multiple sound channels.

Listing 2-24 Defining a data structure to track many sound channels

```
CONST
   kMaxNumSndChans = 20;            {max number of sound channels}
TYPE
   SCInfo =
   RECORD
      sndChan:       SndChannelPtr; {NIL or pointer to channel}
      mustDispose:   Boolean;       {flag to dispose channel}
      itsData:       Handle;        {data to dispose with channel}
   END;
   SCList = ARRAY[1..kMaxNumSndChans] OF SCInfo;
VAR
   gSndChans:        SCList;
```

The `SCInfo` data structure defined in Listing 2-24 allows you to keep track of which channels in the collection are being used and which were being used but currently need disposal; it also allows you to associate data with a sound channel so that you can dispose of the data when you dispose of the sound channel. Note that the value of the `kMaxNumSndChans` constant might vary from application to application. Having defined the data structure, you must initialize it (so that the `sndChan` and `itsData` fields are `NIL` and the `mustDispose` field is `FALSE`). You must also write a procedure that finds an available channel. You might declare such a procedure like this:

```
PROCEDURE DoTrackChan (chanToTrack: SndChannelPtr; associatedData: Handle);
```

Using such a procedure, you could simply create sound channels by using local variables and then add them to the tracking list so that your application disposes of them when they finish executing. The exact implementation of such a procedure would depend on the needs of your application. For example, if there are no channels available in the global list of sound channels, your application might report an error, stop sound on all active channels, or stop sound on the channel that has been playing the longest. If you want your application to be compatible with computers that do not support

multichannel sound, this procedure could check whether multichannel sound is supported, and if not, would stop any sound playing on other channels. This is particularly useful if your application plays sound effects in response to actions on the screen; overlapping sound effects sound best, but if this is unattainable, the newest sound should have the highest priority.

One advantage of maintaining a list of sound channels is that you can use it in conjunction with both callback procedures and completion routines. Listing 2-25 defines a procedure that either your callback procedure or completion routine could call after setting the application's A5 world correctly.

Listing 2-25 Marking a channel for disposal

```
PROCEDURE MySetTrackChanDispose (mySndChannel: SndChannelPtr);
VAR
   index:      Integer;       {channel index}
   found:      Boolean;       {flag variable}
BEGIN
   index := 1;                {start at first spot}
   found := FALSE;            {initialize flag variable}
   WHILE (index <= kMaxNumSndChans) AND (NOT found) DO
      IF gSndChans[index].sndChan = mySndChannel THEN
         found := TRUE        {proper channel found}
      ELSE
         index := index + 1;  {move to next spot}
   IF found THEN
      gSndChans[index].mustDispose := TRUE;
END;
```

The final thing you need to do is to define a procedure that your application calls once each time through its main event loop. This procedure must dispose of sound channels that are marked for disposal. Listing 2-26 defines such a routine.

Listing 2-26 Disposing of channels that have been marked for disposal

```
PROCEDURE MyCleanUpTrackedChans;
CONST
   kQuietNow = TRUE;                          {need to quiet channel?}
VAR
   index:      Integer;
   myErr:      OSErr;
BEGIN
   FOR index := 1 TO kMaxNumSndChans DO       {go through all channels}
   WITH gSndChans[index] DO
```

```
      IF mustDispose THEN                          {check global flag}
      BEGIN                                        {channel needs disposal}
         IF gSndChans[index].itsData <> NIL THEN
         BEGIN                                     {release other data}
            HUnlock(gSndChans[index].itsData);
            HPurge(gSndChans[index].itsData);
         END;
                                                   {free channel-related memory}
         myErr := MyDisposeSndChannel(sndChan, kQuietNow);
         sndChan := NIL;                           {set pointer to NIL}
         mustDispose := FALSE;                     {reset global flag}
         IF myErr <> noErr THEN
            DoError(myErr);
      END;
END;
```

The `MyCleanUpTrackedChans` procedure defined in Listing 2-26 works just like the `MyCheckSndChan` procedure defined in Listing 2-20, but instead of checking a single global flag, it checks the flag associated with each allocated sound channel. Now that you have defined such a procedure, you can easily write a routine to stop sound in all active channels (for example, if your application receives a suspend event). Simply set the `mustDispose` flag on all sound channels that are allocated (that is for all channels that are not `NIL`) and then call `MyCleanUpTrackedChans`. Note, however, that when the `MyCleanUpTrackedChans` procedure disposes of a sound channel processing a play from disk, the completion routine will be called and will thus set the `mustDispose` flag to `TRUE`. Thus, the `mustDispose` flag must be reset to `FALSE` *after* the sound channel has been disposed. Otherwise, the `MyCleanUpTrackedChans` procedure would try to dispose of the same sound channel again when the application called it from its main event loop.

Parsing Sound Resources and Sound Files

This section explains how you can parse sound resources and sound files to find the component of a sound resource or sound file that contains information about the sound. For sound resources, this information is stored in the sound header. In addition to obtaining information about a sound from a sound header, you might need a pointer to a sound header to use any of several low-level sound commands. For sound files, information is stored in the Form and Common Chunks. This section shows how you can find those chunks and extract information from them.

Note

The techniques shown in this section assume that you are familiar with the format of sound resources and sound files. See “Sound Storage Formats” beginning on page 2-73 for complete information on sound storage formats. ◆

Obtaining a Pointer to a Sound Header

This section shows how you can obtain a pointer to a sound header stored in a sound resource. You can use this pointer to obtain information about the sound. You also need a pointer to a sound header to install a sampled sound as a voice in a channel (as described in "Installing Voices Into Channels" on page 2-43) and to play sounds using low-level sound commands (as described below and in the next section). You can use a technique similar to the one described in this section if you wish to obtain a pointer to wave-table data that is stored in a sound resource.

Sound Manager versions 3.0 and later include the `GetSoundHeaderOffset` function that you can use to locate a sound header embedded in a sound resource. Listing 2-27 shows how to call the `GetSoundHeaderOffset` function and then pass the returned offset to the `bufferCmd` sound command, to play a sampled sound using low-level Sound Manager routines.

Listing 2-27 Playing a sound resource

```
FUNCTION MyPlaySampledSound (chan: SndChannelPtr; sndHandle: Handle): OSErr;
VAR
   myOffset:      LongInt;
   mySndCmd:      SndCommand;          {a sound command}
   myErr:         OSErr;
BEGIN
   myErr := GetSoundHeaderOffset(sndHandle, myOffset);
   IF myErr = noErr THEN
   BEGIN
      HLock(sndHandle);
      mySndCmd.cmd := bufferCmd;               {command is bufferCmd}
      mySndCmd.param1 := 0;                    {unused with bufferCmd}
      mySndCmd.param2 := LongInt(ORD4(sndHandle^) + myOffset);
      myErr := SndDoImmediate(chan, mySndCmd);
   END;
   MyPlaySampledSound := myErr;
END;
```

If the `GetSoundHeaderOffset` function is not available but you still need to obtain a pointer to a sound header, you can use the function `MyGetSoundHeaderOffset` defined in Listing 2-28. The function defined there traverses a sound resource until it reaches the sound data. It returns, in the `offset` parameter, the offset in bytes from the beginning of a sound resource to the sound header.

IMPORTANT

The `GetSoundHeaderOffset` function is available in Sound Manager versions 3.0 and later. As a result, you'll need to use the techniques illustrated in Listing 2-28 only if you want your application to find a sound header when earlier versions of the Sound Manager are available. ▲

Listing 2-28 Obtaining the offset in bytes to a sound header

```
FUNCTION MyGetSoundHeaderOffset (sndHdl: Handle; VAR offset: LongInt): OSErr;
TYPE
   Snd1Header =                          {format 1 'snd ' resource header}
   RECORD
      format:     Integer;               {format of resource}
      numSynths:  Integer;               {number of data types}
                                         {synths, init option follow}
   END;
   Snd1HdrPtr = ^Snd1Header;
   Snd2Header =                          {format 2 'snd ' resource header}
   RECORD
      format:     Integer;               {format of resource}
      refCount:   Integer;               {for application use}
   END;
   Snd2HdrPtr = ^Snd2Header;
   IntPtr = ^Integer;                    {for type coercion}
   SndCmdPtr = ^SndCommand;              {for type coercion}
VAR
   myPtr:         Ptr;                   {to navigate resource}
   myOffset:      LongInt;               {offset into resource}
   numSynths:     Integer;               {info about resource}
   numCmds:       Integer;               {info about resource}
   isDone:        Boolean;               {are we done yet?}
   myErr:         OSErr;
BEGIN
   {Initialize variables.}
   myOffset := 0;                        {return 0 if no sound header found}
   myPtr := Ptr(sndHdl^);                {point to start of resource data}
   isDone := FALSE;                      {haven't yet found sound header}
   myErr := noErr;

   {Skip everything before sound commands.}
   CASE Snd1HdrPtr(myPtr)^.format OF
      firstSoundFormat:                  {format 1 'snd ' resource}
         BEGIN                           {skip header start, synth ID, etc.}
```

```
            numSynths := Snd1HdrPtr(myPtr)^.numSynths;
            myPtr := Ptr(ORD4(myPtr) + SizeOf(Snd1Header));
            myPtr := Ptr(ORD4(myPtr) +
                        numSynths * (SizeOf(Integer) + SizeOf(LongInt)));
         END;
      secondSoundFormat:                  {format 2 'snd ' resource}
         myPtr := Ptr(ORD4(myPtr) + SizeOf(Snd2Header));
      OTHERWISE                           {unrecognized resource format}
         BEGIN
            myErr := badFormat;
            isDone := TRUE;
         END;
      END;

   {Find number of commands and move to start of first command.}
   numCmds := IntPtr(myPtr)^;
   myPtr := Ptr(ORD4(myPtr) + SizeOf(Integer));

   {Search for bufferCmd or soundCmd to obtain sound header.}
   WHILE (numCmds >= 1) AND (NOT isDone) DO
   BEGIN
      IF (IntPtr(myPtr)^ = bufferCmd + dataOffsetFlag) OR
            (IntPtr(myPtr)^ = soundCmd + dataOffsetFlag) THEN
      BEGIN                               {bufferCmd or soundCmd found}
                                          {copy offset from sound command}
         myOffset := SndCmdPtr(myPtr)^.param2;
         isDone := TRUE;                  {get out of loop}
      END
      ELSE
      BEGIN                               {soundCmd or bufferCmd not found}
                                          {move to next command}
         myPtr := Ptr(ORD4(myPtr) + SizeOf(SndCommand));
         numCmds := numCmds - 1;
      END;
   END; {WHILE}

   offset := myOffset;                    {return offset}
   MyGetSoundHeaderOffset := myErr;       {return result code}
END;
```

The `MyGetSoundHeaderOffset` function defined in Listing 2-28 begins by initializing several variables, including a pointer that it sets to point to the beginning of the data contained in the sound resource. Then, after determining whether the sound resource is

format 1 or format 2, the function skips data contained in the format 1 `'snd '` resource header or in the format 2 `'snd '` resource header, as appropriate.

Note
Do not confuse the format 1 or format 2 `'snd '` header with the sound header the `MyGetSoundHeaderOffset` function defined in Listing 2-28 is designed to find. A sound header contains information about the sampled-sound data stored in a sound resource; a sound resource header contains information about the format of the sound resource. ◆

After skipping information in the sound resource header, `MyGetSoundHeaderOffset` simply looks through all sound commands in the resource for a `bufferCmd` or `soundCmd` command, either of which must contain the offset from the beginning of the resource to the sound header in its `param2` field. If the given sound resource contains no sound header (and thus no sampled-sound data), the `MyGetSoundHeaderOffset` function returns an error and sets the `offset` variable parameter to 0.

After using the `MyGetSoundHeaderOffset` function to obtain an offset to the sound header, you can easily obtain a pointer to a sound header. Note, however, that because a handle to a sound resource is contained in a relocatable block, you must lock the relocatable block before you obtain a pointer to a sound header, and you must not unlock it until you are through using the pointer. Listing 2-29 demonstrates how you can convert an offset to a sound header into a pointer to a sound header after locking a relocatable block.

Listing 2-29 Converting an offset to a sound header into a pointer to a sound header

```
FUNCTION MyGetSoundHeader (sndHandle: Handle): SoundHeaderPtr;
VAR
   myOffset:    LongInt;         {offset to sound header}
   myErr:      OSErr;
BEGIN
   HLockHi(sndHandle);           {lock data in high memory}
                                 {compute offset to sound header}
   myErr := MyGetSoundHeaderOffset(sndHandle, myOffset);
   IF myErr <> noErr THEN
      MyGetSoundHeader := NIL    {no sound header in resource}
   ELSE
                                 {compute address of sound header}
      MyGetSoundHeader := SoundHeaderPtr(ORD4(sndHandle^) + myOffset);
END;
```

The `MyGetSoundHeader` function defined in Listing 2-29 locks the sound handle you pass it in high memory and then attempts to find an offset to the sound header in the sound handle. If the `MyGetSoundHeaderOffset` function defined in Listing 2-28 returns an offset of 0, then `MyGetSoundHeader` returns a `NIL` pointer to a sound

header; otherwise, it returns a pointer that remains valid as long as you do not unlock the sound handle.

The `MyGetSoundHeader` function returns a pointer to a sampled sound header even if the sound header is actually an extended sound header or a compressed sound header. Thus, before accessing any other fields of the sound header, you should test the `encode` field of the sound header to determine what type of sound header it is. Then, if the sound header is, for example, an extended sound header, cast the sampled sound header to an extended sound header. Then you can access any of the fields of the extended sound header. For an example of this technique, see Listing 2-16 on page 2-44.

Playing Sounds Using Low-Level Routines

Once you obtain a pointer to a sampled sound header, you can use the `bufferCmd` sound command to play a sound without using the high-level Sound Manager routines. Many sampled-sound resources include `bufferCmd` commands, so the high-level Sound Manager routines often issue the `bufferCmd` command indirectly. Thus, you might in some cases be able to make your application slightly more efficient by issuing the `bufferCmd` command directly. Also, you might issue a `bufferCmd` command directly if you want the Sound Manager to ignore other parts of a sound resource.

Finally, you might issue `bufferCmd` commands directly if you want your application to be able to play a large sound resource without loading the entire resource at once. By issuing several successive `bufferCmd` commands, you can play a large sound resource using a small buffer. In this case, each buffer must contain a sampled sound header. In most cases, the sound will play smoothly, without audible gaps. It's generally easier, however, to play large sampled sounds from disk by using the play-from-disk routines or the `SndPlayDoubleBuffer` function. See "Managing Double Buffers" on page 2-147 for complete details.

Note
Using the `bufferCmd` command to play several consecutive compressed samples on the Macintosh Plus, the Macintosh SE, or the Macintosh Classic is not guaranteed to work without an audible pause or click. ◆

The pointer in the `param2` field of a `bufferCmd` command is the location of a sampled sound header. A `bufferCmd` command is queued in the channel until the preceding commands have been processed. If the `bufferCmd` command is contained within an `'snd '` resource, the high bit of the command must be set. If the sound was loaded in from an `'snd '` resource, your application is expected to unlock this resource and allow it to be purged after using it. Listing 2-30 shows how your application can play a sampled sound stored in a resource using the `bufferCmd` command.

Listing 2-30 Playing a sound using the `bufferCmd` command

```
FUNCTION MyLowLevelSampledSndPlay (chan: SndChannelPtr; sndHandle: Handle):
                                   OSErr;
CONST
   kWaitIfFull = TRUE;                {wait for room in queue?}
VAR
   mySndHeader:   SoundHeaderPtr;
   mySndCmd:      SndCommand;         {a sound command}
BEGIN
   mySndHeader := MyGetSoundHeader(sndHandle);
   WITH mySndCmd DO
   BEGIN
      cmd := bufferCmd;               {command is bufferCmd}
      param1 := 0;                    {unused with bufferCmd}
      param2 := LongInt(mySndHeader);  {pointer to sound header}
   END;
   IF mySndHeader <> NIL THEN
      MyLowLevelSampledSndPlay :=
                              SndDoCommand(chan, mySndCmd, NOT kWaitIfFull)
   ELSE
      MyLowLevelSampledSndPlay := badFormat;
END;
```

For the `MyLowLevelSampledSndPlay` function defined in Listing 2-30 to play a sound, the channel passed to it must already be configured to play sampled-sound data. Otherwise, the function returns a `badChannel` result code. Also, because the `bufferCmd` command works asynchronously, you might want to associate a callback procedure with the sound channel when you create the channel. For more information on playing sounds asynchronously, see "Playing Sounds Asynchronously" on page 2-46.

You can use the `bufferCmd` command to handle compressed sound samples in addition to sounds that are not compressed. To expand and play back a buffer of compressed samples, you pass the Sound Manager a `bufferCmd` command where `param2` points to a compressed sound header.

To play sampled sounds that are not compressed, pass `bufferCmd` a standard or extended sound header. The extended sound header can be used for stereo sampled sounds. The standard sampled sound header is used for all other noncompressed sampled sounds.

Finding a Chunk in a Sound File

Sound files are not as tightly structured as sound resources. As explained in "Sound Files" on page 2-81, the chunks in a sound file can appear in any order, except that the Form Chunk is always first. Most information about a sampled sound stored in a sound file is contained in the Common Chunk. Thus, to be able to access this information, you

must be able to find a particular kind of chunk in a sound file. Listing 2-31 defines a procedure that you can use to find the location of the first chunk of a specified type beginning at the chunk you specify.

IMPORTANT
The techniques illustrated in this section are provided primarily to help you understand the structure of sound files. Most sound-producing applications don't need to parse sound files. ▲

Listing 2-31 Finding a chunk in a sound file

```
FUNCTION MyFindChunk (myFile: Integer;       {file reference number}
                      myChunkSought: ID;     {ID of chunk sought}
                      startPos: LongInt;     {file position to start at}
                      VAR chunkFPos: LongInt) {file position of found chunk}
                      : OSErr;
VAR
   myLength:          LongInt;               {number of bytes to read}
   myChunkHeader:     ChunkHeader;           {characteristics of chunk}
   found:             Boolean;               {flag variable}
   myErr:             OSErr;                 {error from File Manager calls}
BEGIN
   found := FALSE;                           {initialize flag variable}
                                             {set file mark at start}
   myErr := SetFPos(myFile, fsFromStart, startPos);

   {Search file's chunks for desired chunk ID.}
   WHILE (NOT found) AND (myErr = noErr) DO
   BEGIN                                     {check current chunk}
      myLength := SizeOf(myChunkHeader);
      {Load chunk header.}
      myErr := FSRead(myFile, myLength, @myChunkHeader);
      IF myErr = noErr THEN                  {chunk header loaded okay}
         IF myChunkHeader.ckID = myChunkSought THEN
         BEGIN
            found := TRUE;                   {chunk has been found}
                                             {find position in file}
            myErr := GetFPos(myFile, chunkFPos);
                                             {compute chunk's start position}
            chunkFPos := chunkFPos - SizeOf(myChunkHeader);
         END
         ELSE
         BEGIN                               {move to next chunk}
            IF myChunkHeader.ckID = ID(FormID) THEN
```

```
            {Adjust Form Chunk's size to size of formType field.}
            myChunkHeader.ckSize := SizeOf(ID);
          IF myChunkHeader.ckSize MOD 2 = 1 THEN
            {Compensate for pad byte.}
            myChunkHeader.ckSize := myChunkHeader.ckSize + 1;
          myErr := SetFPos(myFile, fsFromMark, myChunkHeader.ckSize);
        END;
   END; {WHILE}
   MyFindChunk := myErr;
END;
```

The `MyFindChunk` function defined in Listing 2-31 accepts four parameters. The `myFile` parameter is the file reference number of an open sound file. (For information on file reference numbers, see *Inside Macintosh: Files*.) In the `myChunkSought` parameter, you pass the ID of the type of chunk you wish to find. For example, you might pass `ID(FormID)` to find the Form Chunk. The third parameter, `startPos`, is the file position at which `MyFindChunk` should start searching for a chunk. This file position must be the beginning of a chunk. To start at the beginning of a file, specify 0. Finally, if the `MyFindChunk` function is successful, it returns in the `chunkFPos` parameter the file position of the first chunk of the specified type that it found. If the function is unsuccessful, it returns the appropriate File Manager result code (such as an end-of-file error) and the `chunkFPos` parameter is undefined.

The `MyFindChunk` function works by looking at each chunk of the sound file, beginning at the file position `startPos` and checking to see if the chunk is of the type sought. If a chunk matches, the `MyFindChunk` function returns the file position of the start of the chunk; otherwise, the function moves onto the next chunk. For each chunk, the `MyFindChunk` function reads in the chunk header, checks for a match, and then moves to the next chunk.

The `MyFindChunk` function moves from one chunk to the next by identifying the size of the current chunk, not including the chunk header, from the `ckSize` field of the chunk header. Whenever you parse sound files, you should always use the `ckSize` field of the chunk header to determine the size of a chunk if the size of the chunk could vary in size. The `MyFindChunk` function adjusts the value in the `ckSize` field before advancing to the next chunk in two cases. First, the `ckSize` field for the Form Chunk reflects the size of the entire sound file, so this function changes it to the size of the `formType` field so that the function does not skip the file's local chunks. Second, if the `ckSize` field is odd, 1 byte is added because the number of bytes in a chunk is always even.

After using the `MyFindChunk` function defined in Listing 2-31, you might still need to read the data contained in a chunk into memory. For example, you might read in the Form and Common Chunks to obtain information about a sound file. Listing 2-32 uses the `MyFindChunk` function to find a chunk in a sound file, allocates an appropriately sized block of memory for that chunk, and reads the chunk into that block.

Listing 2-32 Loading a chunk from a sound file

```
FUNCTION MyGetChunkData (myFile: Integer;       {file reference number}
                         myChunkSought: ID;     {ID of chunk sought}
                         startPos: LongInt):    {file position to start at}
                         Ptr;                   {pointer to data or NIL}
VAR
   myFPos:           LongInt;                   {position in file}
   myLength:         LongInt;                   {number of bytes to read}
   myChunkHeader:    ChunkHeader;               {characteristics of a chunk}
   myChunkData:      Ptr;                       {pointer to chunk data}
   myErr:            OSErr;
BEGIN
   myChunkData := NIL;                          {initialize variable}
   myErr := MyFindChunk(myFile, myChunkSought, startPos, myFPos);
   IF myErr = noErr THEN
                                                {move to start of chunk}
      myErr := SetFPos(myFile, fsFromStart, myFPos);
   IF myErr = noErr THEN
   BEGIN                                {determine how much data to copy}
      myLength := SizeOf(ChunkHeader);
      myErr := FSRead(myFile, myLength, @myChunkHeader);
      IF myChunkHeader.ckID = ID(FormID) THEN
         myChunkHeader.ckSize := SizeOf(ID);    {don't return local chunks}
      myLength := myChunkHeader.ckSize + SizeOf(ChunkHeader);
      IF myErr = noErr THEN
                                                {return to chunk's start}
         myErr := SetFPos(myFile, fsFromStart, myFPos);
   END;
   IF myErr = noErr THEN
   BEGIN                                    {read chunk data into RAM}
      myChunkData := NewPtr(myLength);
      IF myChunkData <> NIL THEN
         myErr := FSRead(myFile, myLength, myChunkData);
   END;
   IF myErr <> noErr THEN
      IF myChunkData <> NIL THEN
         DisposePtr(myChunkData);
   MyGetChunkData := myChunkData;
END;
```

The `MyGetChunkData` function defined in Listing 2-32 attempts to find a chunk in a file. If it finds the chunk, it reads the chunk header to determine the chunk's size, and if the chunk is the Form Chunk, adjusts the chunk size so that the sound file's local chunks are

not included in the chunk size. Then the function attempts to allocate memory for the chunk and read the chunk into the memory. If a problem occurs at any time, the function simply returns `NIL`.

Note
The format of a sound file might not be the same as its operating-system type. In particular, a file might have an operating-system type `'AIFC'` but be formatted as an AIFF file because the sampled-sound data contained in the file is noncompressed. ◆

Compressing and Expanding Sounds

Some of the capabilities provided by MACE are transparently available to your application. For example, if you pass the `SndPlay` function a handle to an `'snd '` resource that contains a compressed sampled sound, the Sound Manager automatically expands the sound data for playback in real time. Your application does not need to know whether the `'snd '` resource contains compressed or noncompressed samples when it calls `SndPlay`. This is because sufficient information is in the resource itself to allow the Sound Manager to determine whether it should expand the data samples.

However, aside from expansion playback, all of the MACE capabilities need to be specifically requested by your application. For example, you can use the procedure `Comp3to1` or `Comp6to1` if you want to compress a sampled sound (for example, to create an `'snd '` resource containing compressed audio data). You can use the procedures `Exp1to3` and `Exp1to6` to expand compressed audio data.

All of these procedures require you to specify both an input and an output buffer, from and to which the sampled-sound data to be converted is read and written. Your application must allocate the appropriate amount of storage for each buffer. For example, if you want to expand a buffer of compressed monophonic sampled-sound data by using `Exp1to6`, the output buffer must be at least six times the size of the input buffer.

The MACE compression and expansion routines can work on only one channel of sound. The `numChannels` parameter of all four procedures allows you to specify how many channels are in the original sample, and the `whichChannel` parameter allows you to specify which channel you wish to compress or expand. Because the MACE routines can compress or expand only one channel of sound, you must make adjustments when allocating an output buffer for stereo sound. For example, if you are compressing two-channel sound using the `Comp3to1` procedure, your output buffer need only be one-sixth the size of your input buffer.

Often when compressing polyphonic sound, being able to compress only one channel is not a problem, because you lose sound quality during compression anyway. However, you might at times wish to maintain more than one channel of a multichannel sound even after compression and expansion. For example, two channels of a stereo sound might be quite different and might both be necessary to achieve a full sound after expansion. In these cases, you can compress each channel of a multichannel sound individually and then manually interleave the samples on a packet basis. When you

expand polyphonic compressed sound data, you must interleave the channels of sound on a sample frame basis.

The MACE routines work only with sampled-sound data in offset binary format. If you are compressing data in a sound file, you must convert that data from linear, two's complement format to binary offset format before compression.

When calling the MACE routines, you can also specify addresses of two small buffers (128 bytes each) that the Sound Manager uses to maintain state information about the compression or expansion process. When you first call a MACE routine, the state buffers should be filled with zeros to initialize the state information. When you subsequently call another MACE routine, you can use the same state buffers. You can pass `NIL` for both buffers if you do not want to save state information across calls to the MACE routines. Listing 2-33 illustrates the use of the `Comp3to1` procedure when using state buffers.

Listing 2-33 Compressing audio data

```
PROCEDURE MyCompressBy3 (inBuf: Ptr; outBuf: Ptr; numSamp: LongInt);
CONST
   kStateBufferSize = 128;
VAR
   myInState:        Ptr;     {input state buffer}
   myOutState:       Ptr;     {output state buffer}
BEGIN
   myInState := NewPtrClear(kStateBufferSize);
   myOutState := NewPtrClear(kStateBufferSize);
   IF (myInState <> NIL) AND (myOutState <> NIL) THEN
      Comp3to1(inBuf, outBuf, numSamp, myInState, myOutState, 1, 1);
END;
```

Because the last two parameters (`numChannels` and `whichChannel`) are both set to 1, `MyCompressBy3` compresses monophonic audio data.

In practice, compressing a sound resource or sound file is considerably more complex than calling the `MyCompressBy3` procedure defined in Listing 2-33. To compress a sound resource containing monophonic sampled-sound data, you would need to

- load the data into a handle and lock the handle
- ensure that the data in the handle is not already compressed by examining the sound header
- find a pointer to the sampled-sound data by examining the `samplePtr` field of the sound header
- allocate an output buffer of the appropriate size, taking into account that only one channel of the original data can be compressed
- compress the sampled-sound data by calling the `Comp3To1` procedure

- determine the size that the header information (including, for example, sound commands and the sampled sound header excluding the sampled-sound data itself) will take in the resource by using the Sound Input Manager's `SetupSndHeader` function to create a sound resource header and sampled sound header with the same sample rate, base frequency, and other characteristics as the original sampled-sound data
- resize the handle so that it is large enough to contain both the non–sampled-sound data information and the compressed sound data
- fill this handle by first calling `SetupSndHeader` once again and by then copying the compressed sound data to the end of the header information
- update the resource file

Techniques for compressing sound files and for expanding both sound resources and sound files are analogous to that sketched here. Remember that after compressing or expanding each channel of polyphonic sampled-sound data, you must interleave frames of sound data, on a packet basis after compression or on a sample basis after expansion.

Using Double Buffers

The play-from-disk routines make extensive use of the `SndPlayDoubleBuffer` function. You can use this function in your application directly if you wish to bypass the normal play-from-disk routines. You might want to do this to maximize the efficiency of your application while maintaining compatibility with the Sound Manager. Or, you might define your own double-buffering routines so that your application can convert 16-bit sound data on disk to 8-bit data that all versions of the Sound Manager can play. By using `SndPlayDoubleBuffer` instead of the normal play-from-disk routines, you can specify your own doubleback procedure (that is, the algorithm used to switch back and forth between buffers) and customize several other buffering parameters.

IMPORTANT

`SndPlayDoubleBuffer` is a very low-level routine and is not intended for general use. In most cases, you should use the high-level Sound Manager routines (such as `SndPlay` or `SndStartFilePlay`) or standard sound commands (such as `bufferCmd`) to play sounds. You should use `SndPlayDoubleBuffer` only if you require very fine control over double buffering. Remember also that the `SndPlayDoubleBuffer` function is not always available. You'll need to ensure that it's available in the current operating environment before calling it. See "Testing for Multichannel Sound and Play-From-Disk Capabilities" beginning on page 2-35 for details. ▲

You call `SndPlayDoubleBuffer` by passing it a pointer to a sound channel (into which the double-buffered data is to be written) and a pointer to a sound double buffer header record. Here's an example:

```
myErr := SndPlayDoubleBuffer(mySndChan, @myDoubleHeader);
```

A sound double buffer header record has the following structure:

```
TYPE SndDoubleBufferHeader =
PACKED RECORD
   dbhNumChannels:   Integer;    {number of sound channels}
   dbhSampleSize:    Integer;    {sample size, if noncompressed}
   dbhCompressionID: Integer;    {ID of compression algorithm}
   dbhPacketSize:    Integer;    {number of bits per packet}
   dbhSampleRate:    Fixed;      {sample rate}
   dbhBufferPtr:     ARRAY[0..1] OF SndDoubleBufferPtr;
                                 {pointers to SndDoubleBuffer}
   dbhDoubleBack:    ProcPtr;    {pointer to doubleback procedure}
END;
```

The values for the `dbhCompressionID`, `dbhNumChannels`, and `dbhPacketSize` fields are the same as those for the `compressionID`, `numChannels`, and `packetSize` fields of the compressed sound header, respectively.

The `dbhBufferPtr` array contains pointers to two records of type `SndDoubleBuffer`. These are the two buffers between which the Sound Manager switches until all the sound data has been sent into the sound channel. When the call to `SndPlayDoubleBuffer` is made, the two buffers should both already contain a nonzero number of frames of data.

IMPORTANT

The Sound Manager defines the data type `SndDoubleBufferHeader2` that is identical to the `SndDoubleBufferHeader` data type except that it contains the `dbhFormat` field (of type `OSType`) that defines a custom codec to be used to decompress the sound data. The `dbhFormat` field is used only if the `dbhCompressionID` field contains the value `fixedCompression`. See "Sound Double Buffer Header Records" beginning on page 2-111 for details. ▲

Here is the structure of a sound double buffer:

```
TYPE SndDoubleBuffer =
PACKED RECORD
   dbNumFrames:   LongInt;       {number of frames in buffer}
   dbFlags:       LongInt;       {buffer status flags}
   dbUserInfo:    ARRAY[0..1] OF LongInt;
                                 {for application's use}
   dbSoundData:   PACKED ARRAY[0..0] OF Byte;
                                 {array of data}
END;
```

The buffer status flags field for each of the two buffers might contain either of these values:

```
CONST
   dbBufferReady     = $00000001;
   dbLastBuffer      = $00000004;
```

All other bits in the `dbFlags` field are reserved by Apple; your application should not modify them.

The following two sections illustrate how to fill out these data structures, create your two buffers, and define a doubleback procedure to refill the buffers when they become empty.

Setting Up Double Buffers

Before you can call `SndPlayDoubleBuffer`, you need to allocate two buffers (of type `SndDoubleBuffer`), fill them both with data, set the flags for the two buffers to `dbBufferReady`, and then fill out a record of type `SndDoubleBufferHeader` with the appropriate information. Listing 2-34 illustrates how you can accomplish these tasks.

Listing 2-34 Setting up double buffers

```
CONST
   kDoubleBufferSize = 4096;     {size of each buffer (in bytes)}
TYPE
   LocalVars =                   {variables used by the doubleback procedure}
   RECORD
      bytesTotal:    LongInt;    {total number of samples}
      bytesCopied:   LongInt;    {number of samples copied to buffers}
      dataPtr:       Ptr;        {pointer to sample to copy}
   END;
   LocalVarsPtr = ^LocalVars;

{This function uses SndPlayDoubleBuffer to play the sound specified.}
FUNCTION MyDBSndPlay (chan: SndChannelPtr; sndHeader: SoundHeaderPtr): OSErr;
VAR
   myVars:           LocalVars;
   myDblHeader:      SndDoubleBufferHeader;
   myDblBuffer:      SndDoubleBufferPtr;
   myStatus:         SCStatus;
   myIndex:          Integer;
   myErr:            OSErr;
BEGIN
   {Set up myVars with initial information.}
   myVars.bytesTotal := sndHeader^.length;
   myVars.bytesCopied := 0;              {no samples copied yet}
   myVars.dataPtr := Ptr(@sndHeader^.sampleArea[0]);
```

```
                                        {pointer to first sample}
{Set up SndDoubleBufferHeader.}
WITH myDblHeader DO
BEGIN
   dbhNumChannels := 1;                 {one channel}
   dbhSampleSize := 8;                  {8-bit samples}
   dbhCompressionID := 0;               {no compression}
   dbhPacketSize := 0;                  {no compression}
   dbhSampleRate := sndHeader^.sampleRate;
   dbhDoubleBack := @MyDoubleBackProc;
END;

FOR myIndex := 0 TO 1 DO                {initialize both buffers}
BEGIN
   {Get memory for double buffer.}
   myDblBuffer := SndDoubleBufferPtr(NewPtr(Sizeof(SndDoubleBuffer) +
                                            kDoubleBufferSize));
   IF myDblBuffer = NIL THEN
   BEGIN
      MyDBSndPlay := MemError;
      Exit(MyDBSndPlay);
   END;

   myDblBuffer^.dbNumFrames := 0;       {no frames yet}
   myDblBuffer^.dbFlags := 0;           {buffer is empty}
   myDblBuffer^.dbUserInfo[0] := LongInt(@myVars);

   {Fill buffer with samples.}
   MyDoubleBackProc(sndChan, myDblBuffer);

   {Store buffer pointer in header.}
   myDblHeader.dbhBufferPtr[myIndex] := myDblBuffer;
END;
{Start the sound playing.}
myErr := SndPlayDoubleBuffer(sndChan, @myDblHeader);
IF myErr <> noErr THEN
BEGIN
   MyDBSndPlay := myErr;
   Exit(MyDBSndPlay);
END;

{Wait for the sound's end by checking the channel status.}
REPEAT
```

```
      myErr := SndChannelStatus(chan, sizeof(myStatus), @status);
   UNTIL NOT myStatus.scChannelBusy;

   {Dispose double buffer memory.}
   FOR myIndex := 0 TO 1 DO
      DisposePtr(Ptr(myDblHeader.dbhBufferPtr[myIndex]));

   MyDBSndPlay := noErr;
END;
```

The function `MyDBSndPlay` takes two parameters, a pointer to a sound channel and a pointer to a sound header. For information about obtaining a pointer to a sound header, see "Obtaining a Pointer to a Sound Header" on page 2-57. The `MyDBSndPlay` function reads the sound header to determine the characteristics of the sound to be played (for example, how many samples are to be sent into the sound channel). Then `MyDBSndPlay` fills in the fields of the double buffer header, creates two buffers, and starts the sound playing. The doubleback procedure `MyDoubleBackProc` is defined in the next section.

Writing a Doubleback Procedure

The `dbhDoubleBack` field of a double buffer header specifies the address of a doubleback procedure, an application-defined procedure that is called when the double buffers are switched and the exhausted buffer needs to be refilled. The doubleback procedure should have this format:

```
PROCEDURE MyDoubleBackProc (chan: SndChannelPtr;
                            exhaustedBuffer: SndDoubleBufferPtr);
```

The primary responsibility of the doubleback procedure is to refill an exhausted buffer of samples and to mark the newly filled buffer as ready for processing. Listing 2-35 illustrates how to define a doubleback procedure. Note that the sound channel pointer passed to the doubleback procedure is not used in this procedure.

This doubleback procedure extracts the address of its local variables from the `dbUserInfo` field of the double buffer record passed to it. These variables are used to keep track of how many total bytes need to be copied and how many bytes have been copied so far. Then the procedure copies at most a bufferfull of bytes into the empty buffer and updates several fields in the double buffer record and in the structure containing the local variables. Finally, if all the bytes to be copied have been copied, the buffer is marked as the last buffer.

Note

Because the doubleback procedure is called at interrupt time, it cannot make any calls that move memory either directly or indirectly. (Despite its name, the `BlockMove` procedure does not cause blocks of memory to move or be purged, so you can safely call it in your doubleback procedure, as illustrated in Listing 2-35.) ◆

Listing 2-35 Defining a doubleback procedure

```
PROCEDURE MyDoubleBackProc (chan: SndChannelPtr;
                            doubleBuffer: SndDoubleBufferPtr);
VAR
   myVarsPtr:        LocalVarsPtr;
   myNumBytes:       LongInt;
BEGIN
   {Get pointer to my local variables.}
   myVarsPtr := LocalVarsPtr(doubleBuffer^.dbUserInfo[0]);

   {Get number of bytes left to copy.}
   myNumBytes := myVarsPtr^.bytesTotal - myVarsPtr^.bytesCopied;

   {If the amount left is greater than double buffer size, limit the number }
   { of bytes to copy to the size of the buffer.}
   IF myNumBytes > kDoubleBufferSize THEN
      myNumBytes := kDoubleBufferSize;

   {Copy samples to double buffer.}
   BlockMove(myVarsPtr^.dataPtr, @doubleBuffer^.dbSoundData[0], myNumBytes);

   {Store number of samples in buffer and mark buffer as ready.}
   doubleBuffer^.dbNumFrames := myNumBytes;
   doubleBuffer^.dbFlags := BOR(doubleBuffer^.dbFlags, dbBufferReady);

   {Update data pointer and number of bytes copied.}
   myVarsPtr^.dataPtr := Ptr(ORD4(myVarsPtr^.dataPtr) + myNumBytes);
   myVarsPtr^.bytesCopied := myVarsPtr^.bytesCopied + myNumBytes;

   {If all samples have been copied, then this is the last buffer.}
   IF myVarsPtr^.bytesCopied = myVarsPtr^.bytesTotal THEN
      doubleBuffer^.dbFlags := BOR(doubleBuffer^.dbFlags, dbLastBuffer);
END;
```

Sound Storage Formats

This section describes in detail the formats of sound resources and sound files, which are the two principal storage formats for sound data on Macintosh computers. In general, an application that uses the services provided by the Sound Manager and the Sound Input Manager to play and record sounds does not need to know how the sound data is

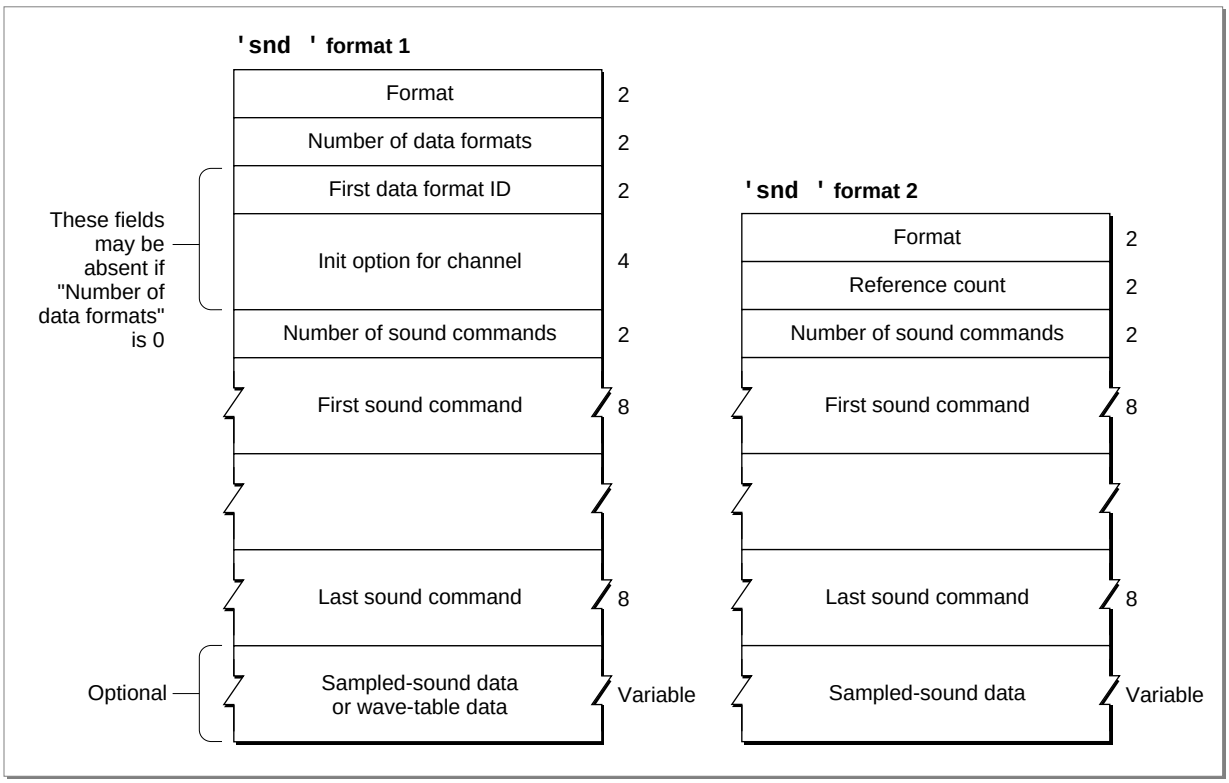
organized in memory or on disk. For some special purposes, however, you might need the information in this section.

Sound Resources

A **sound resource** is a resource of type `'snd '` that contains sound commands and possibly also sound data. Sound resources are widely used by Macintosh applications that produce sounds. These resources provide a simple and portable way for you to incorporate sounds into your application. For example, the sounds that a user can select in the Sound control panel as the system alert sound are stored in the System file as `'snd '` resources.

There are two types of `'snd '` resources, known as format 1 and format 2. Figure 2-4 illustrates the structures of both kinds of `'snd '` resources.

Figure 2-4 The structure of `'snd '` resources



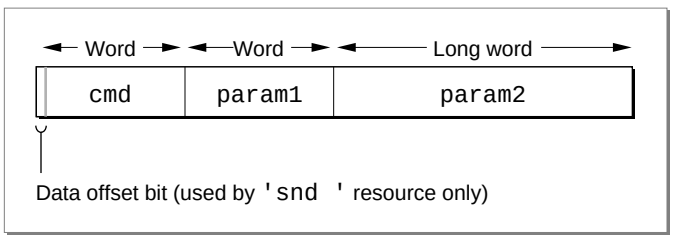
IMPORTANT

The format 2 `'snd '` resource is obsolete. Your application should create only format 1 `'snd '` resources. The format 2 `'snd '` resource was designed for use by HyperCard and can be used with sampled-sound data only. ▲

Resource IDs for `'snd '` resources in the range 0 to 8191 are reserved for use by Apple Computer, Inc. The `'snd '` resources numbered 1 through 4 are defined to be the standard system alert sounds, although more recent versions of system software have included more standard system alert sounds.

When a sound command contained in an `'snd '` resource has associated sound data, the high bit of the command is set. This changes the meaning of the `param2` field of the command from a pointer to a location in RAM to an offset value that specifies the offset in bytes from the resource's beginning to the location of the associated sound data (such as a sampled sound header). Figure 2-5 illustrates the location of this data offset bit.

Figure 2-5 The location of the data offset bit



The offset bit is used only by sound commands that are stored in sound resources of type `'snd '` and that have associated sound data (that is, sampled-sound or wave-table data).

You can use a constant to access that flag.

```
CONST
   dataOffsetFlag    = $8000;  {sound command data offset bit}
```

If the `dataOffsetFlag` bit is not set, `param2` is interpreted instead as a pointer to the location in memory (outside the sound resource) where the data is located.

The first few bytes of the resource contain `'snd '` header information and are a different size for each format. An audio data type specified in a format 1 `'snd '` requires 6 bytes. The number of data types multiplied by 6 is added to this offset. The number of commands multiplied by 8 bytes, the size of a sound command, is added to the offset.

The Format 1 Sound Resource

Figure 2-4 shows the fields of a format 1 `'snd '` resource. A format 1 `'snd '` resource header contains information about the format of the resource (namely, 1), the data type, and the initialization options for that data type. A format 1 `'snd '` resource contains sound commands and might also contain the actual sound data for wave-table sounds or sampled sounds. Note that if a sound resource includes sampled-sound data, then part of the sound data section is devoted to a sound header that describes the sampled-sound data in the remainder of the sound data section.

If an 'snd ' resource specifies a data type, it can supply an initialization option in the field immediately following the type. You specify the number of commands in the resource in the number of sound commands field. The sound commands follow, in the order in which they should be sent to the sound channel.

The format 1 'snd ' resource might contain only a sequence of commands describing a sound. In this case, the number of data types should be 0, and there should be no data type specification or initialization option in the 'snd ' resource. This allows the 'snd ' resource to be used with any kind of sound data.

Listing 2-36 shows the output of the MPW tool DeRez when applied to the 'snd ' resource with resource ID 1 contained in the System file.

Listing 2-36 A format 1 'snd ' resource

```
data 'snd ' (1, "Simple Beep", purgeable) {
   /*the sound resource header*/
   $"0001"     /*format type*/
   $"0001"     /*number of data types*/
   $"0001"     /*square-wave data*/
   $"00000000" /*initialization option*/
   /*the sound commands*/
   $"001B"     /*number of sound commands (27)*/
   $"002C"     /*command 1--timbreCmd 090 000*/
   $"005A00000000"
   $"002B"     /*command 2--ampCmd  224 000*/
   $"00E000000000"
   $"002A"     /*command 3--freqCmd 000 069*/
   $"000000000045"
   $"000A"     /*command 4--waitCmd 040 000*/
   $"002800000000"
   $"002B"     /*command 5--ampCmd  200 000*/
   $"00C800000000"
   /*commands 6 through 26 are omitted; they are */
   /* alternating pairs of waitCmd and ampCmd commands */
   /* where the first parameter of ampCmd has the */
   /* values 192, 184, 176, 168, 160, 144, 128, 96, */
   /* 64, and 32*/
   $"002B"     /*command 27--ampCmd  000 000*/
   $"000000000000"
};
```

As you can see, the Simple Beep is actually a rather sophisticated sound, in which the loudness (or amplitude) of the beep gradually decreases from an initial value of 224 to 0.

Notice that the sound shown in Listing 2-36 is defined using square-wave data and is completely determined by a sequence of specific commands. (“Play an A at loudness 224, wait 20 milliseconds, play it at loudness 200....”) Often, an `'snd '` resource consists only of a single sound command (usually the `bufferCmd` command) together with data that describes a sampled sound to be played. Listing 2-37 shows an example like this.

Listing 2-37 A format 1 `'snd '` resource containing sampled-sound data

```
data 'snd ' (19068, "hello daddy", purgeable) {
   /*the sound resource header*/
   $"0001"        /*format type*/
   $"0001"        /*number of data types*/
   $"0005"        /*sampled-sound data*/
   $"00000080"    /*initialization option: initMono*/
   /*the sound commands*/
   $"0001"        /*number of sound commands that follow (1)*/
   $"8051"        /*command 1--bufferCmd*/
   $"0000"        /*param1 = 0*/
   $"00000014"    /*param2 = offset to sound header (20 bytes)*/
   /*the sampled sound header*/
   $"00000000"    /*pointer to data (it follows immediately)*/
   $"00000BB8"    /*number of bytes in sample (3000 bytes)*/
   $"56EE8BA3"    /*sampling rate of this sound (22 kHz)*/
   $"000007D0"    /*starting of the sample's loop point*/
   $"00000898"    /*ending of the sample's loop point*/
   $"00"          /*standard sample encoding*/
   $"3C"          /*baseFrequency at which sample was taken*/
                  /*the sampled-sound data*/
   $"80 80 81 81 81 81 81 81 80 80 80 80 80 81 82 82"
   $"82 83 82 82 81 80 80 7F 7F 7F 7E 7D 7D 7D 7C 7C"
   $"7C 7C 7D 7D 7D 7D 7E 7F 80 80 81 81 82 82 83 83"
   $"83 83 82 81 81 80 80 81 81 81 81 81 82 81 81 80"
   $"80 80 81 81 81 83 83 83 82 81 81 80 7F 7E 7D 7D"
   $"7F 7F 7F 7F 7E 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 80"
   /*rest of data omitted in this example*/
};
```

This `'snd '` resource indicates that the sound is defined using sampled-sound data. The resource includes a call to a single sound command, the `bufferCmd` command. The offset bit of the command number is set to indicate that the sound data is contained in the resource itself. Following the command and its two parameters is the sampled sound header, the first part of which contains important information about the sample. The second parameter to the `bufferCmd` command indicates the offset from the beginning of the resource to the sampled sound header, in this case 20 bytes. After the sound

commands, this resource includes a sampled sound header, which includes the sampled-sound data. The format of a sampled sound header is described in "Sound Header Records" on page 2-104.

For compressed sound data, the sampled sound header is replaced by a compressed sampled sound header. Listing 2-38 illustrates the structure of an `'snd '` resource that contains compressed sound data.

Listing 2-38 An `'snd '` resource containing compressed sound data

```
data 'snd ' (9004, "Raisa's Cry", purgeable) {
   /*the sound resource header*/
   $"0001"     /*format type*/
   $"0001"     /*number of data types*/
   $"0005"     /*first data type*/
   $"00000380" /*initialization option: initMACE3 + initMono*/
   /*the sound command*/
   $"0001"     /*number of sound commands that follow (1)*/
   $"8051"     /*cmd: bufferCmd*/
   $"0000"     /*param1: unused*/
   $"00000014" /*param2: offset to sound header (20 bytes)*/
   /*the compressed sampled sound header*/
   $"00000000" /*pointer to data (it follows immediately)*/
   $"00000001" /*number of channels in sample*/
   $"56EE8BA3" /*sampling rate of this sound (22 kHz)*/
   $"00000000" /*starting of the sample's loop point; not used*/
   $"00000000" /*ending of the sample's loop point; not used*/
   $"FE"       /*compressed sample encoding*/
   $"00"       /*baseFrequency; not used*/
   $"00006590" /*number of frames in sample (26,000)*/
   $"400DADDD1745D145826B"
               /*AIFFSampleRate (22 kHz in extended type)*/
   $"00000000" /*markerChunk; NIL for 'snd ' resource*/
   $"4D414333" /*format; MACE 3:1 compression*/
   $"00000000" /*futureUse2; NIL for 'snd ' resource*/
   $"00000000" /*stateVars; NIL for 'snd ' resource*/
   $"00000000" /*leftOverBlockPtr; not used here*/
   $"FFFF"     /*compressionID, -1 means use format field*/
   $"0010"     /*packetSize, packetSize for 3:1 is 16 bits*/
   $"0000"     /*snthID is 0*/
   $"0008"     /*sampleSize, sound was 8-bit before processing*/
   $"2F 85 81 32 64 87 33 86"    /*the compressed sound data*/
   $"6F 48 6D 65 72 6B 82 88"
   $"91 FE 8D 8E 86 4E 7C E9"
```

```
   $"6F 6D 71 70 7E 79 4F 83"
   $"59 8F 8F 65" /*rest of data omitted in this example*/
};
```

This resource has the same general structure as the `'snd '` resource illustrated in Listing 2-36. The principal difference is that the standard sound header is replaced by the compressed sound header. This example resource specifies a monophonic sound compressed by using the 3:1 compression algorithm. A multichannel compressed sound's data would be interleaved on a packet basis. See "Compressed Sound Header Records" beginning on page 2-108 for a complete explanation of the compressed sound header.

As you've seen, it is not always necessary to specify `'snd '` resources by listing the raw data stream contained in them; indeed, for certain types of format 1 `'snd '` resources, it can be easier to supply a resource specification like the one given in Listing 2-39.

Listing 2-39 A resource specification

```
resource 'snd ' (9000, "Nathan's Beep", purgeable) {
   FormatOne {
      {  /*array of data types: 1 element*/
      /*[1]*/
      squareWaveSynth, 0
      }
   },
   {     /*array SoundCmnds: 3 elements*/
   /*[1]*/ noData, timbreCmd {90},
   /*[2]*/ noData, freqDurationCmd {480, $00000045},
   /*[3]*/ noData, quietCmd {},
      },
   {     /*array DataTables: 0 elements*/
   };
};
```

When you pass a handle to this resource to the `SndPlay` function, three commands are executed by the Sound Manager: a `timbreCmd` command, a `freqDurationCmd` command, and a `quietCmd` command. The sound specified in Listing 2-39 is just like the Simple Beep, except that there is no gradual reduction in the loudness. Listing 2-40 shows a resource specification for the Simple Beep.

Listing 2-40 A resource specification for the Simple Beep

```
resource 'snd ' (9001, "Copy of Simple Beep", purgeable) {
   FormatOne {
      {  /*array of data types: 1 element*/
```

```
            /*[1]*/
            squareWaveSynth, 0
         }
      },
      {  /*array SoundCmnds: 27 elements*/
         /*[1]*/     nodata, timbreCmd {90},
         /*[2]*/     nodata, ampCmd {224},
         /*[3]*/     nodata, freqCmd {69},
         /*[4]*/     nodata, waitCmd {40},
         /*[5]*/     nodata, ampCmd {200},
         /*[6]*/     nodata, waitCmd {40},
         /*[7]*/     nodata, ampCmd {192},
         /*[8]*/     nodata, waitCmd {40},
         /*[9]*/     nodata, ampCmd {184},
         /*[10]*/    nodata, waitCmd {40},
         /*[11]*/    nodata, ampCmd {176},
         /*[12]*/    nodata, waitCmd {40},
         /*[13]*/    nodata, ampCmd {168},
         /*[14]*/    nodata, waitCmd {40},
         /*[15]*/    nodata, ampCmd {160},
         /*[16]*/    nodata, waitCmd {40},
         /*[17]*/    nodata, ampCmd {144},
         /*[18]*/    nodata, waitCmd {40},
         /*[19]*/    nodata, ampCmd {128},
         /*[20]*/    nodata, waitCmd {40},
         /*[21]*/    nodata, ampCmd {96},
         /*[22]*/    nodata, waitCmd {40},
         /*[23]*/    nodata, ampCmd {64},
         /*[24]*/    nodata, waitCmd {40},
         /*[25]*/    nodata, ampCmd {32},
         /*[26]*/    nodata, waitCmd {40},
         /*[27]*/    nodata, ampCmd {0},
      },
      {  /*array DataTables: 0 elements*/
      }
};
```

The Format 2 Sound Resource

The `SndPlay` function can also play format 2 `'snd '` resources, which are designed for use only with sampled sounds. The `SndPlay` function supports this format by automatically opening a sound channel and using the `bufferCmd` command to send the data contained in the resource to the channel.

Figure 2-4 illustrates the fields of a format 2 `'snd '` resource. The reference count field is for your application's use and is not used by the Sound Manager. The number of sound commands field and the sound command fields are the same as described in a format 1 resource. The last field of this resource contains the sampled sound. The first command should be either a `soundCmd` command or `bufferCmd` command with the data offset bit set in the command to specify the location of this sampled sound header.

Listing 2-41 shows a resource specification that illustrates the structure of a format 2 `'snd '` resource.

Listing 2-41 A format 2 `'snd '` resource

```
data 'snd ' (9003, "Pig Squeal", purgeable) {
   /*the sound resource header*/
   $"0002"        /*format type*/
   $"0000"        /*reference count for application's use*/
   /*the sound command*/
   $"0001"        /*number of sound commands that follow (1)*/
   $"8051"        /*command 1--bufferCmd*/
   $"0000"        /*param1 = 0*/
   $"0000000E"    /*param2 = offset to sound header (14 bytes)*/
   /*the sampled sound header*/
   $"00000000"    /*pointer to data (it follows immediately)*/
   $"00000BB8"    /*number of bytes in sample (3000 bytes)*/
   $"56EE8BA3"    /*sampling rate of this sound (22 kHz)*/
   $"000007D0"    /*starting of the sample's loop point*/
   $"00000898"    /*ending of the sample's loop point*/
   $"00"          /*standard sample encoding*/
   $"3C"          /*baseFrequency at which sample was taken*/
   $"80 80 81 82 84 87 93 84"    /*the sampled-sound data*/
   $"6F 68 6D 65 72 7B 82 88"
   $"91 8E 8D 8F 86 7E 7C 79"
   $"6F 6D 71 70 70 79 7F 81"
   $"89 8F 8D 8B" /*rest of data omitted in this example*/
};
```

Note
Remember that format 2 `'snd '` resources are obsolete. You should create only format 1 `'snd '` resources. ◆

Sound Files

This section describes in detail the structure of AIFF and AIFF-C files. Both of these types of sound files are collections of **chunks** that define characteristics of the sampled sound or other relevant data about the sound.

Note
Most applications only need to read AIFF and AIFF-C files or to record sampled-sound data directly to them. You can both play and record AIFF and AIFF-C files without knowing the details of the AIFF and AIFF-C file formats, as explained in the chapter "Introduction to Sound on the Macintosh" in this book. Thus, the information in this section is for advanced programmers only. ◆

Currently, the AIFF and AIFF-C specifications include the following chunk types.

Chunk type	Description
Form Chunk	Contains information about the format of an AIFF or AIFF-C file and contains all the other chunks of such a file.
Format Version Chunk	Contains an indication of the version of the AIFF-C specification according to which this file is structured (AIFF-C only).
Common Chunk	Contains information about the sampled sound such as the sampling rate and sample size.
Sound Data Chunk	Contains the sample frames that comprise the sampled sound.
Marker Chunk	Contains markers that point to positions in the sound data.
Comments Chunk	Contains comments about markers in the file.
Sound Accelerator Chunk	Contains information intended to allow applications to accelerate the decompression of compressed audio data.
Instrument Chunk	Defines basic parameters that an instrument (such as a sampling keyboard) can use to play back the sound data.
MIDI Data Chunk	Contains MIDI data.
Audio Recording Chunk	Contains information pertaining to audio recording devices.
Application Specific Chunk	Contains application-specific information.
Name Chunk	Contains the name of the sampled sound.
Author Chunk	Contains one or more names of the authors (or creators) of the sampled sound.
Copyright Chunk	Contains a copyright notice for the sampled sound.
Annotation Chunk	Contains a comment.

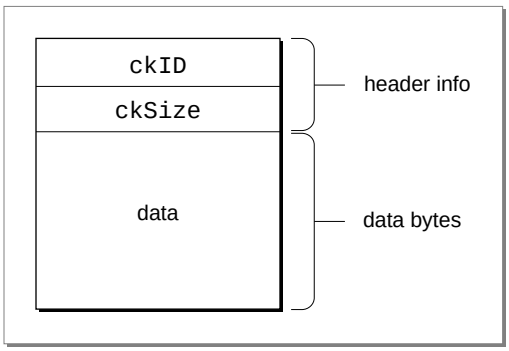
The following sections document the four principal kinds of chunks that can occur in AIFF and AIFF-C files.

Chunk Organization and Data Types

An AIFF or AIFF-C file contains several different types of chunks. For example, there is a Common Chunk that specifies important parameters of the sampled sound, such as its size and sample rate. There is also a Sound Data Chunk that contains the actual audio samples. A chunk consists of some header information followed by some data. The

header information consists of a chunk ID number and a number that indicates the size of the chunk data. In general, therefore, a chunk has the structure shown in Figure 2-6.

Figure 2-6 The general structure of a chunk



The header information of a chunk has this structure:

```
TYPE ChunkHeader =
RECORD
   ckID:    ID;         {chunk type ID}
   ckSize:  LongInt;    {number of bytes of data}
END;
```

The `ckID` field specifies the chunk type. An ID is a 32-bit concatenation of any four printable ASCII characters in the range `' '` (space character, ASCII value $20) through `'~'` (ASCII value $7E). Spaces cannot precede printing characters, but trailing spaces are allowed. Control characters are not allowed. You can specify values for the four types of chunks described later by using these constants:

```
CONST
   FormID           = 'FORM';   {ID for Form Chunk}
   FormatVersionID  = 'FVER';   {ID for Format Version Chunk}
   CommonID         = 'COMM';   {ID for Common Chunk}
   SoundDataID      = 'SSND';   {ID for Sound Data Chunk}
```

The `ckSize` field specifies the size of the data portion of a chunk and does not include the length of the chunk header information.

The Form Chunk

The chunks that define the characteristics of a sampled sound and that contain the actual sound data are grouped together into a container chunk, known as the Form Chunk. The Form Chunk defines the type and size of the file and holds all remaining chunks in the file. The chunk ID for this container chunk is `'FORM'`.

A chunk of type 'FORM' has this structure:

```
TYPE ContainerChunk =
RECORD
   ckID:       ID;         {'FORM'}
   ckSize:     LongInt;    {number of bytes of data}
   formType:   ID;         {type of file}
END;
```

For a Form Chunk, the ckSize field contains the size of the data portion of this chunk. Note that the data portion of a Form Chunk is divided into two parts, formType and the rest of the chunks of the file, which follow the formType field. These chunks are called *local chunks* because their chunk IDs are local to the Form Chunk.

The local chunks can occur in any order in a sound file. As a result, your application should be designed to get a local chunk, identify it, and then process it without making any assumptions about what kind of chunk it is based on its order in the Form Chunk.

The formType field of the Form Chunk specifies the format of the file. For AIFF files, formType is 'AIFF'. For AIFF-C files, formType is 'AIFC'. Note that this type might not be the same as the operating-system type with which the File Manager identifies the file. In particular, a file of operating-system type 'AIFC' might be formatted as an AIFF file.

The Format Version Chunk

One difference between the AIFF and AIFF-C file formats is that files of type AIFF-C contain a Format Version Chunk and files of type AIFF do not. The Format Version Chunk contains a timestamp field that indicates when the format version of this AIFF-C file was defined. This in turn indicates what format rules this file conforms to and allows you to ensure that your application can handle a particular AIFF-C file. Every AIFF-C file must contain one and only one Format Version Chunk.

In AIFF-C files, a Format Version Chunk has this structure:

```
TYPE FormatVersionChunk =
RECORD
   ckID:       ID;         {'FVER'}
   ckSize:     LongInt;    {4}
   timestamp:  LongInt;    {date of format version}
END;
```

Note
In AIFF files, there is no Format Version Chunk. ◆

The timestamp field indicates when the format version for this kind of file was created. The value indicates the number of seconds since January 1, 1904, following the normal time conventions used by the Macintosh Operating System. (See the chapter on date and

time utilities in *Inside Macintosh: Operating System Utilities* for several routines that allow you to manipulate time stamps.)

You should not confuse the format version time stamp with the creation date of the file. The format version time stamp indicates the time of creation of the version of the format according to which this file is structured. Because Apple defines the formats of AIFF-C files, only Apple can change this value. The current version is defined by a constant:

```
CONST
   AIFCVersion1   = $A2805140;   {May 23, 1990, 2:40 p.m.}
```


The Common Chunk

Every AIFF and AIFF-C file must contain a Common Chunk that defines some fundamental characteristics of the sampled sound contained in the file. Note that the format of the Common Chunk is different for AIFF and AIFF-C files. As a result, you need to determine the type of file format (by inspecting the `formType` field of the Form Chunk) before reading the Common Chunk.

For AIFF files, the Common Chunk has this structure:

```
TYPE CommonChunk =
RECORD
   ckID:             ID;         {'COMM'}
   ckSize:           LongInt;    {size of chunk data}
   numChannels:      Integer;    {number of channels}
   numSampleFrames:  LongInt;    {number of sample frames}
   sampleSize:       Integer;    {number of bits per sample}
   sampleRate:       Extended;   {number of frames per second}
END;
```

For AIFF-C files, the Common Chunk has this structure:

```
TYPE ExtCommonChunk =
RECORD
   ckID:             ID;         {'COMM'}
   ckSize:           LongInt;    {size of chunk data}
   numChannels:      Integer;    {number of channels}
   numSampleFrames:  LongInt;    {number of sample frames}
   sampleSize:       Integer;    {number of bits per sample}
   sampleRate:       Extended;   {number of frames per second}
   compressionType:  ID;         {compression type ID}
   compressionName:  PACKED ARRAY[0..0] OF Byte;
                                 {compression type name}
END;
```

The fields that exist in both types of Common Chunk have the following meanings:

The numChannels field of both types of Common Chunk indicate the number of audio channels contained in the sampled sound. A value of 1 indicates monophonic sound, a value of 2 indicates stereo sound, a value of 4 indicates four-channel sound, and so forth. Any number of audio channels may be specified. The actual sound data is stored elsewhere, in the Sound Data Chunk.

The numSampleFrames field indicates the number of sample frames in the Sound Data Chunk. Note that this field contains the number of sample frames, not the number of bytes of data and not the number of sample points. For noncompressed sound data, the total number of sample points in the file is numChannels * numSampleFrames. (For more information on sample points, see "Sampled-Sound Data" on page 2-9.)

The sampleSize field indicates the number of bits in each sample point of noncompressed sound. Although the field can contain any integer from 1 to 32, the Sound Manager currently supports only 8- and 16-bit sound. For compressed sound data, this field indicates the number of bits per sample in the original sound data, before compression.

The sampleRate field contains the sample rate at which the sound is to be played back, in sample frames per second. For a list of common sample rates, see Table 2-1 on page 2-16.

An AIFF-C Common Chunk includes two fields that describe the type of compression (if any) used on the audio data. The compressionType field contains the type of the compression algorithm, if any, used on the sound data. Here are the currently available compression types and their associated compression names:

```
CONST
   {compression types}
   NoneType                = 'NONE';
   ACE2Type                = 'ACE2';
   ACE8Type                = 'ACE8';
   MACE3Type               = 'MAC3';
   MACE6Type               = 'MAC6';
```

You can define your own compression types, but you should register them with Apple.

Finally, the compressionName field contains a human-readable name for the compression algorithm ID specified in the compressionType field. Compression names for Apple-supplied codecs are defined by constants:

```
CONST
   {compression names}
   NoneName                = 'not compressed';
   ACE2to1Name             = 'ACE 2-to-1';
   ACE8to3Name             = 'ACE 8-to-3';
   MACE3to1Name            = 'MACE 3-to-1';
   MACE6to1Name            = 'MACE 6-to-1';
```

This string is useful when putting up alert boxes (perhaps because a necessary decompression routine is missing). Pad the end of this array with a byte having the value 0 if the length of this array is not an even number (but do not include the pad byte in the count).

The Sound Data Chunk

The Sound Data Chunk contains the actual sample frames that make up the sampled sound. The Sound Data Chunk has this structure:

```
TYPE SoundDataChunk =
RECORD
   ckID:       ID;         {'SSND'}
   ckSize:     LongInt;    {size of chunk data}
   offset:     LongInt;    {offset to sound data}
   blockSize:  LongInt;    {size of alignment blocks}
END;
```

The `offset` field indicates an offset (in bytes) to the beginning of the first sample frame in the chunk data. Most applications do not need to use the `offset` field and should set it to 0.

The `blockSize` field contains the size (in bytes) of the blocks to which the sound data is aligned. This field is used in conjunction with the `offset` field for aligning sound data to blocks. As with the `offset` field, most applications do not need to use the `blockSize` field and should set it to 0.

The sampled-sound data follows the `blockSize` field. For information on the format of sampled-sound data, see "Sampled-Sound Data" on page 2-9.

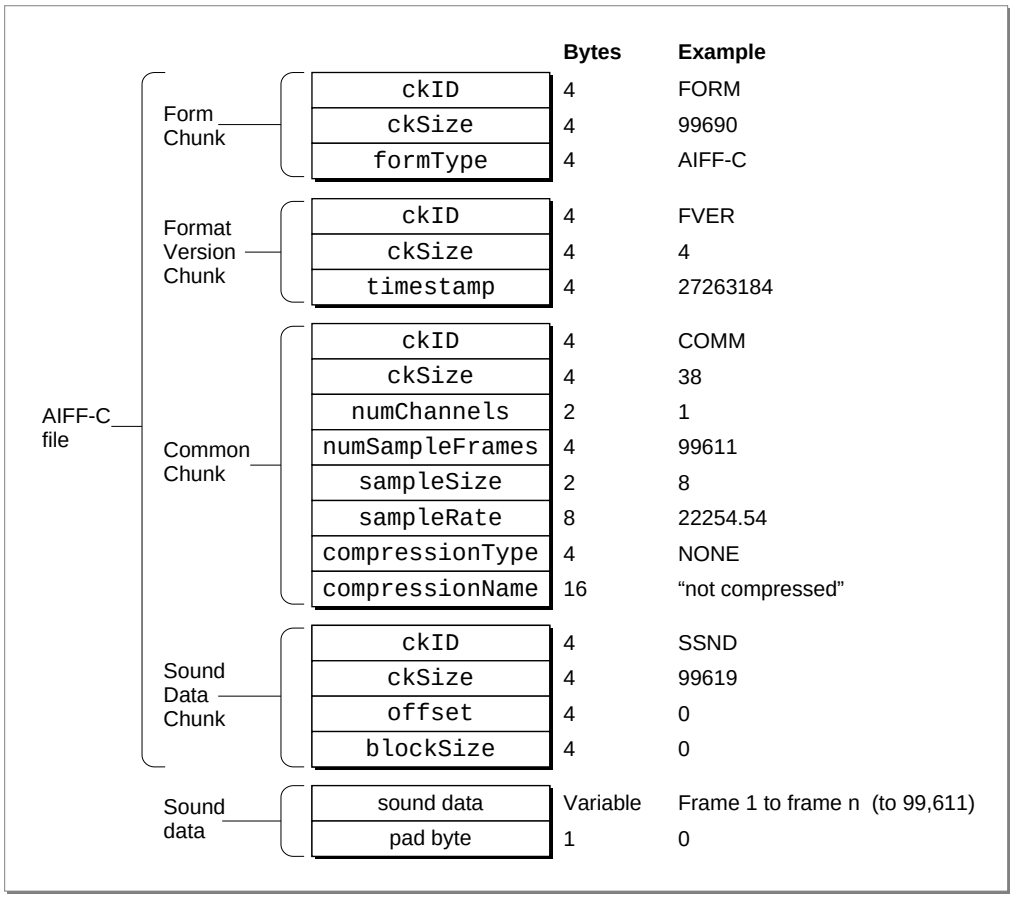
Note
The Sound Data Chunk is required unless the `numSampleFrames` field in the Common Chunk is 0. A maximum of one Sound Data Chunk can appear in an AIFF or AIFF-C file. ◆

Format of Entire Sound Files

Figure 2-7 illustrates an AIFF-C file that contains approximately 4.476 seconds of 8-bit monophonic sound data sampled at 22 kHz. The sound data is not compressed. Note that the number of sample frames in this example is odd, forcing a pad byte to be inserted after the sound data. This pad byte is not reflected in the `ckSize` field of the Sound Data Chunk, which means that special processing is required to correctly determine the actual chunk size.

On a Macintosh computer, the Form Chunk (and hence all the other chunks in an AIFF or AIFF-C file) is stored in the data fork of the file. The file type of an AIFF format file is `'AIFF'`, and the file type of an AIFF-C format file is `'AIFC'`. Macintosh applications should not store any information in the resource fork of an AIFF or AIFF-C file because that information might not be preserved by other applications that edit sound files.

Figure 2-7 A sample AIFF-C file



Every Form Chunk must contain a Common Chunk, and every AIFF-C file must contain a Format Version Chunk. In addition, if the sampled sound has a length greater than 0, there must be a Sound Data Chunk in the Form Chunk. All other chunk types are optional. Your application should be able to read all the required chunks if it uses AIFF or AIFF-C files, but it can choose to ignore any of the optional chunks.

When reading AIFF or AIFF-C files, you should keep the following points in mind:

- Remember that the local chunks in an AIFF or AIFF-C file can occur in any order. An application that reads these types of files should be designed to get a chunk, identify it, and then process it without making any assumptions about what kind of chunk it is based on its order.
- If your application allows modification of a chunk, then it must also update other chunks that might be based on the modified chunk. However, if there are chunks in the file that your application does not recognize, you must discard those unrecognized chunks. Of course, if your application is simply copying the AIFF or AIFF-C file without any modification, you should copy the unrecognized chunks, too.

- You can get the clearest indication of the number of sample frames contained in an AIFF or AIFF-C file from the `numSampleFrames` parameter in the Common Chunk, not from the `ckSize` parameter in the Sound Data Chunk. The `ckSize` parameter is padded to include the fields that follow it, but it does not include the byte with a value of 0 at the end if the total number of sound data bytes is odd.
- Remember that each chunk must contain an even number of bytes. Chunks whose total contents would yield an odd number of bytes must have a pad byte with a value of 0 added at the end of the chunk. This pad byte is not included in the `ckSize` field.
- Remember that the `ckSize` field of any chunk does not include the first 8 bytes of the chunk (which specify the chunk type).

Sound Manager Reference

This section describes the constants, data structures, and routines provided by the Sound Manager. It also describes the format of data stored in sound resources and files that the Sound Manager can play.

The section "Constants" describes the constants defined by the Sound Manager that you can use to specify channel initialization parameters and sound commands. It also lists the sound attributes selector for the `Gestalt` function and the returned bit numbers. See the section "Summary of the Sound Manager" on page 2-157 for a list of all the constants defined by the Sound Manager.

The section "Data Structures" beginning on page 2-99 describes the Pascal data structures for all of the Sound Manager records that applications can use, including sound commands, sound channels, and sound headers.

The section "Sound Manager Routines" beginning on page 2-119 describes the routines that allow you to play sounds, manage sound channels, and obtain sound-related information. That section also includes information on routines that give you low-level control over sound output.

The section "Application-Defined Routines" beginning on page 2-151 describes callback procedures and completion routines that your application might need to define.

The section "Resources" beginning on page 2-154 describes the organization of format 1 and format 2 `'snd '` resources.

Constants

This section describes the constants that you can use to specify channel initialization parameters, sound commands, and chunk IDs. It also lists the `Gestalt` function sound attributes selector and the returned bit numbers. All other constants defined by the Sound Manager are described at the appropriate location in this chapter. (For example, the constants that you can use to specify sound data types are described in connection with the `SndNewChannel` function beginning on page 2-127.)

Gestalt Selector and Response Bits

You can pass the `gestaltSoundAttr` selector to the `Gestalt` function to determine information about the sound capabilities of a Macintosh computer.

```
CONST
   gestaltSoundAttr           = 'snd ';   {sound attributes selector}
```

The `Gestalt` function returns information by setting or clearing bits in the `response` parameter. The bits currently used are defined by constants. Note that most of these bits provide information about the built-in hardware only.

IMPORTANT

Bits 7 through 12 are not defined for versions of the Sound Manager prior to version 3.0. ▲

```
CONST
   gestaltStereoCapability    = 0;     {built-in hw can play stereo sounds}
   gestaltStereoMixing        = 1;     {built-in hw mixes stereo to mono}
   gestaltSoundIOMgrPresent   = 3;     {sound input routines available}
   gestaltBuiltInSoundInput   = 4;     {built-in input hw available}
   gestaltHasSoundInputDevice = 5;     {sound input device available}
   gestaltPlayAndRecord       = 6;     {built-in hw can play while recording}
   gestalt16BitSoundIO        = 7;     {built-in hw can handle 16-bit data}
   gestaltStereoInput         = 8;     {built-in hw can record stereo sounds}
   gestaltLineLevelInput      = 9;     {built-in input hw needs line level}
   gestaltSndPlayDoubleBuffer = 10;    {play from disk routines available}
   gestaltMultiChannels       = 11;    {multiple channels of sound supported}
   gestalt16BitAudioSupport   = 12;    {16-bit audio data supported}
```

Constant descriptions

`gestaltStereoCapability`
Set if the built-in sound hardware is able to produce stereo sounds.

`gestaltStereoMixing`
Set if the built-in sound hardware mixes both left and right channels of stereo sound into a single audio signal for the internal speaker.

`gestaltSoundIOMgrPresent`
Set if the Sound Input Manager is available.

`gestaltBuiltInSoundInput`
Set if a built-in sound input device is available.

`gestaltHasSoundInputDevice`
Set if a sound input device is available. This device can be either built-in or external.

`gestaltPlayAndRecord`
Set if the built-in sound hardware is able to play and record sounds simultaneously. If this bit is clear, the built-in sound hardware can either play or record, but not do both at once. This bit is valid only if

the `gestaltBuiltInSoundInput` bit is set, and it applies only to any built-in sound input and output hardware.

`gestalt16BitSoundIO`
Set if the built-in sound hardware is able to play and record 16-bit samples. This indicates that built-in hardware necessary to handle 16-bit data is available.

`gestaltStereoInput`
Set if the built-in sound hardware can record stereo sounds.

`gestaltLineLevelInput`
Set if the built-in sound input port requires line level input.

`gestaltSndPlayDoubleBuffer`
Set if the Sound Manager supports the play-from-disk routines.

`gestaltMultiChannels`
Set if the Sound Manager supports multiple channels of sound.

`gestalt16BitAudioSupport`
Set if the Sound Manager can handle 16-bit audio data. This indicates that software necessary to handle 16-bit data is available.

Note
For complete information about the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. ◆

Channel Initialization Parameters

You can use the following constants to specify initialization parameters for a sound channel. You need to specify initialization parameters when you call `SndNewChannel`.

```
CONST
   initChanLeft            = $0002;    {left stereo channel}
   initChanRight           = $0003;    {right stereo channel}
   waveInitChannel0        = $0004;    {wave-table channel 0}
   waveInitChannel1        = $0005;    {wave-table channel 1}
   waveInitChanne12        = $0006;    {wave-table channel 2}
   waveInitChannel3        = $0007;    {wave-table channel 3}
   initMono                = $0080;    {monophonic channel}
   initStereo              = $00C0;    {stereo channel}
   initMACE3               = $0300;    {3:1 compression}
   initMACE6               = $0400;    {6:1 compression}
   initNoInterp            = $0004;    {no linear interpolation}
   initNoDrop              = $0008;    {no drop-sample conversion}
```

Constant descriptions

`initChanLeft` Play sounds through the left channel of the Macintosh audio jack.

`initChanRight` Play sounds through the right channel of the Macintosh audio jack.

`waveInitChannel0`
Play sounds through the first wave-table channel.

`waveInitChannel1`
Play sounds through the second wave-table channel.

`waveInitChannel2`
Play sounds through the third wave-table channel.

`waveInitChannel3`
Play sounds through the fourth wave-table channel.

`initMono` Play the same sound through both channels of the Macintosh audio jack and the internal speaker. This is the default channel mode.

`initStereo` Play stereo sounds through both channels of the Macintosh audio jack and the internal speaker. Note that some machines cannot play stereo sounds.

`initMACE3` Assume that the sounds to be played through the channel are MACE 3:1 compressed. The `SndNewChannel` function uses this information to help determine whether it can allocate a new sound channel. A noncompressed sound plays normally, even through a channel that has been initialized for MACE.

`initMACE6` Assume that the sounds to be played through the channel are MACE 6:1 compressed. The `SndNewChannel` function uses this information to help determine whether it can allocate a new sound channel. A noncompressed sound plays normally, even through a channel that has been initialized for MACE.

`initNoInterp` Do not use linear interpolation to smooth a sound played back at a different sample rate from the sound's recorded sample rate. Using the `initNoInterp` initialization parameter decreases the CPU load for this channel. Sounds most affected by the absence of linear interpolation are sinusoidal sounds. Sounds least affected are noisy sound effects like explosions and screams.

`initNoDrop` Do not use **drop-sample conversion** to fake sample rate conversion. Using the `initNoDrop` initialization parameter increases the CPU load for the channel but results in a smoother sound.

The Sound Manager also recognizes the following masks, which you can use to select various channel attributes:

```
CONST
   initPanMask       = $0003;    {mask for right/left pan values}
   initSRateMask     = $0030;    {mask for sample rate values}
   initStereoMask    = $00C0;    {mask for mono/stereo values}
   initCompMask      = $FF00;    {mask for compression IDs}
```

Sound Command Numbers

You can perform many sound-related operations by sending sound commands to a sound channel. For example, to change the volume of a sound that is currently playing, you can send the `ampCmd` sound command to the channel using the `SndDoImmediate`

routine. Similarly, to change the volume of all sounds subsequently to be played in a sound channel, you can send the `volumeCmd` sound command to that channel using the `SndDoCommand` routine.

The `cmd` field of the `SndCommand` data structure (described on page 2-99) specifies the sound command you want to execute. The `param1` and `param2` fields of that structure contain any additional information that might be needed to complete the command. One or both of these parameter fields might be ignored by a particular sound command. In some cases, the Sound Manager returns information to your application in one of the parameter fields.

IMPORTANT

In general, you'll use either `SndDoCommand` or `SndDoImmediate` to send sound commands to a sound channel. With several commands, however, you must use the `SndControl` function to issue the sound command. In Sound Manager version 3.0 and later, however, you virtually never need to use `SndControl` because the commands that require it are either no longer supported (for example, `availableCmd`, `totalLoadCmd`, and `loadCmd`) or are obsolete (for example, `versionCmd`). The sound commands specific to the `SndControl` function are documented here for completeness only. ▲

The sound commands available to your application are defined by constants.

```
CONST
   nullCmd           = 0;        {do nothing}
   quietCmd          = 3;        {stop a sound that is playing}
   flushCmd          = 4;        {flush a sound channel}
   reInitCmd         = 5;        {reinitialize a sound channel}
   waitCmd           = 10;       {suspend processing in a channel}
   pauseCmd          = 11;       {pause processing in a channel}
   resumeCmd         = 12;       {resume processing in a channel}
   callBackCmd       = 13;       {execute a callback procedure}
   syncCmd           = 14;       {synchronize channels}
   availableCmd      = 24;       {see if initialization options are }
                                 { supported}
   versionCmd        = 25;       {determine version}
   totalLoadCmd      = 26;       {report total CPU load}
   loadCmd           = 27;       {report CPU load for a new channel}
   freqDurationCmd   = 40;       {play a note for a duration}
   restCmd           = 41;       {rest a channel for a duration}
   freqCmd           = 42;       {change the pitch of a sound
   ampCmd            = 43;       {change the amplitude of a sound}
   timbreCmd         = 44;       {change the timbre of a sound}
   getAmpCmd         = 45;       {get the amplitude of a sound}
   volumeCmd         = 46;       {set volume}
   getVolumeCmd      = 47;       {get volume}
```

```
waveTableCmd      = 60;       {install a wave table as a voice}
soundCmd          = 80;       {install a sampled sound as a voice}
bufferCmd         = 81;       {play a sampled sound}
rateCmd           = 82;       {set the pitch of a sampled sound}
getRateCmd        = 85;       {get the pitch of a sampled sound}
```

Constant descriptions

`nullCmd` Do nothing.
`param1`: 0 (ignored on input and output)
`param2`: 0 (ignored on input and output)

`quietCmd` Stop the sound that is currently playing. You should send `quietCmd` by using `SndDoImmediate`.
`param1`: 0 (ignored on input and output)
`param2`: 0 (ignored on input and output)

`flushCmd` Remove all commands currently queued in the specified sound channel. A `flushCmd` command does not affect any sound that is currently in progress. You should send `flushCmd` by using `SndDoImmediate`.
`param1`: 0 (ignored on input and output)
`param2`: 0 (ignored on input and output)

`reInitCmd` Reset the initialization parameters specified in `param2` for the specified channel.
`param1`: 0 (ignored on input and output)
`param2`: initialization parameters

`waitCmd` Suspend further command processing in a channel until the specified duration has elapsed. To achieve sounds longer than 32,767 half-milliseconds, Pascal programmers can pass a negative number in `param1`, in which case the sound plays for 32,767 half-milliseconds plus the absolute value of `param1`.
`param1`: duration in half-milliseconds (0 to 65,565)
`param2`: 0 (ignored on input and output)

`pauseCmd` Pause any further command processing in a channel until `resumeCmd` is received.
`param1`: 0 (ignored on input and output)
`param2`: 0 (ignored on input and output)

`resumeCmd` Resume command processing in a channel that was previously paused by `pauseCmd`.
`param1`: 0 (ignored on input and output)
`param2`: 0 (ignored on input and output)

`callBackCmd` Execute the callback procedure specified as a parameter to the `SndNewChannel` function. Both `param1` and `param2` are application-specific; you can use these two parameters to send data to your callback routine.
`param1`: application-defined
`param2`: application-defined

`syncCmd` Synchronize multiple channels of sound. A `syncCmd` command is held in the specified channel, suspending all further command

processing. The `param2` parameter contains an identifier that is arbitrary. Each time the Sound Manager receives `syncCmd`, it decrements the `count` parameter for each channel having that identifier. When the count for a specific channel reaches 0, command processing in that channel resumes.
`param1`: count
`param2`: identifier

`availableCmd` Return 1 in `param1` if the Sound Manager supports the initialization options specified in `param2` and 0 otherwise. However, the Sound Manager might support certain initialization parameters in general but not on a specific machine. You should send `availableCmd` using the `SndControl` function.
`param1`: 0 on input; result of command on output
`param2`: initialization parameters

`versionCmd` Previously, this command determined which version of a sound data format is available. The result is returned in `param2`. The high word of the result indicates the major revision number, and the low word indicates the minor revision number. For example, version 2.0 of a data format would be returned as $00020000. However, this command is obsolete, and your application should not rely on it. You send `versionCmd` by using the `SndControl` function.
`param1`: 0 (ignored on input and output)
`param2`: 0 on input; version on output

`totalLoadCmd` Previously, this command determined the total CPU load factor for all existing sound activity and for a new sound channel having the initialization parameters specified in `param2`. However, this command is obsolete, and your application should not rely on it. You send `totalLoadCmd` by using the `SndControl` function.
`param1`: 0 on input, load factor on output
`param2`: initialization parameters

`loadCmd` Previously, this command determined the CPU load factor that would be incurred by a new channel of sound having the initialization parameters specified in `param2`. The load factor returned in `param1` is the percentage of CPU processing power that the specified sound channel would require. However, this command is obsolete, and your application should not rely on it. You send `loadCmd` by using the `SndControl` function.
`param1`: 0 on input, load factor on output
`param2`: initialization parameters

`freqDurationCmd`
Play the note specified in `param2` for the duration specified in `param1`. To achieve sounds longer than 32,767 half-milliseconds, Pascal programmers can pass a negative number in `param1`, in which case the sound plays for 32,767 half-milliseconds plus the absolute value of `param1`. The `param2` parameter must contain a value in the range 0 to 127. If you want the note to stop playing after the duration specified in `param1`, you must send `quietCmd` after `freqDurationCmd`.

	`param1`: duration in half-milliseconds (0 to 65,565) `param2`: desired frequency
`restCmd`	Rest a channel for a specified duration. The duration is specified in half-milliseconds in `param1`. To achieve sounds longer than 32,767 half-milliseconds, Pascal programmers can pass a negative number in `param1`, in which case the sound plays for 32,767 half-milliseconds plus the absolute value of `param1`. `param1`: duration in half-milliseconds (0 to 65,565) `param2`: 0 (ignored on input and output)
`freqCmd`	Change the frequency (or pitch) of a sound. If no sound is currently playing, then `freqCmd` causes the Sound Manager to begin playing indefinitely at the frequency specified in `param2`. If, however, no instrument is installed in the channel and you attempt to play either wave-table or sampled-sound data, no sound is produced. The `param2` parameter must contain a value in the range 0 to 127. The `freqCmd` command is identical to the `freqDurationCmd` command, except that no duration is specified to a `freqCmd` command. `param1`: 0 (ignored on input and output) `param2`: desired frequency
`ampCmd`	Change the amplitude (or loudness) of a sound. If no sound is currently playing, then `ampCmd` sets the amplitude of the next sound to be played. You specify the amplitude in `param1`; the amplitude should be an integer in the range 0 to 255. `param1`: desired amplitude `param2`: 0 (ignored on input and output)
`timbreCmd`	Change the timbre (or tone) of a sound currently being defined using square-wave data. A timbre value of 0 produces a clear tone; a timbre value of 254 produces a buzzing tone. You can use `timbreCmd` only for sounds defined using square-wave data. `param1`: desired timbre (0 to 254) `param2`: 0 (ignored on input and output)
`getAmpCmd`	Determine the current amplitude (or loudness) of a sound. The amplitude is returned in an integer variable whose address you pass in `param2` and is in the range 0 to 255. `param1`: 0 (ignored on input and output) `param2`: pointer to amplitude variable
`volumeCmd`	Set the right and left volumes of the specified sound channel to the volumes specified in the high and low words of `param2`. The value $0100 represents full volume, and $0080 represents half volume. You can specify values larger than $0100 to overdrive the volume. For example, setting `param2` to $02000200 sets the volume on both left and right speakers to twice full volume. Note, however, that `volumeCmd` is available only in Sound Manager versions 3.0 and later. `param1`: 0 (ignored on input and output) `param2`: high word is right volume, low word is left volume
`getVolumeCmd`	Get the current right and left volumes of the specified sound channel. The volumes are returned in the high and low words of the

long integer pointed to by `param2`. The value $0100 represents full volume, and $0080 represents half volume. Note, however, that `getVolumeCmd` is available only in Sound Manager versions 3.0 and later.
`param1`: 0 (ignored on input and output)
`param2`: pointer to volume data

`waveTableCmd` Install a wave table as a voice in the specified channel. The `param1` parameter specifies the length of the wave table, and the `param2` parameter is a pointer to the wave-table data itself. You can use `waveTableCmd` only for sounds defined using wave-table data.
`param1`: length of wave table
`param2`: pointer to wave-table data

`soundCmd` Install a sampled sound as a voice in a channel. If the high bit of the command is set, `param2` is interpreted as an offset from the beginning of the `'snd '` resource containing the command to the sound header. If the high bit is not set, `param2` is interpreted as a pointer to the sound header. You can use the `soundCmd` command only with noncompressed sampled-sound data. You can also use `soundCmd` to preconfigure a sound channel, so that you can later send sound commands to it at interrupt time.
`param1`: 0 (ignored on input and output)
`param2`: offset or pointer to sound header

`bufferCmd` Play a buffer of sampled-sound data. If the high bit of the command is set, `param2` is interpreted as an offset from the beginning of the `'snd '` resource containing the command to the sound header. If the high bit is not set, `param2` is interpreted as a pointer to the sound header. You can use `bufferCmd` only with sampled-sound data. Note that sending a `bufferCmd` resets the rate of the channel to 1.0.
`param1`: 0 (ignored on input and output)
`param2`: offset or pointer to sound header

`rateCmd` Set the rate of a sampled sound that is currently playing, thus effectively altering its pitch and duration. Your application can set a rate of 0 to pause a sampled sound that is playing. The new rate is set to the value specified in `param2`, which is interpreted relative to 22 kHz. (For example, to set the rate to 44 kHz, pass $00020000 in `param2`; see Listing 2-4 on page 2-26 for sample code that uses `rateCmd`.) You can use `rateCmd` only with sampled-sound data.
`param1`: 0 (ignored on input and output)
`param2`: desired rate of sound

`getRateCmd` Determine the sample rate of the sampled sound currently playing. The current rate of the channel is returned in a `Fixed` variable whose address you pass in `param2` of the sound command. The values returned are always relative to the 22 kHz sampling rate, as with the `rateCmd` sound command. You can use `getRateCmd` only with sampled-sound data, and you should send it by using `SndDoImmediate`.
`param1`: 0 (ignored on input and output)
`param2`: pointer to rate variable

Chunk IDs

You can use the following constants to specify a chunk ID, a 4-byte value that identifies the type of a chunk in an AIFF or AIFF-C file.

```
CONST
   {IDs for AIFF and AIFF-C file chunks}
   FormID                  = 'FORM';   {ID for Form Chunk}
   FormatVersionID         = 'FVER';   {ID for Format Version Chunk}
   CommonID                = 'COMM';   {ID for Common Chunk}
   SoundDataID             = 'SSND';   {ID for Sound Data Chunk}
   MarkerID                = 'MARK';   {ID for Marker Chunk}
   InstrumentID            = 'INST';   {ID for Instrument Chunk}
   MIDIDataID              = 'MIDI';   {ID for MIDI Data Chunk}
   AudioRecordingID        = 'AESD';   {ID for Recording Chunk}
    ApplicationSpecificID  = 'APPL';   {ID for Application Chunk}
   CommentID               = 'COMT';   {ID for Comment Chunk}
   NameID                  = 'NAME';   {ID for Name Chunk}
   AuthorID                = 'AUTH';   {ID for Author Chunk}
   CopyrightID             = '(c) ';   {ID for Copyright Chunk}
   AnnotationID            = 'ANNO';   {ID for Annotation Chunk}
```

Constant descriptions

`FormID` The Form Chunk. A Form Chunk contains information about the format of the file, and contains all the other chunks of the file.

`FormatVersionID` The Format Version Chunk. A Format Version Chunk contains an indication of the version of the AIFF-C specification according to which this file is structured (AIFF-C only).

`CommonID` The Common Chunk. A Common Chunk contains information about the sampled sound, such as the sampling rate and sample size.

`SoundDataID` The Sound Data Chunk. A Sound Data Chunk contains the sample frames that comprise the sampled sound.

`MarkerID` The Marker Chunk. A Marker Chunk contains markers that point to positions in the sound data.

`InstrumentID` The Instrument Chunk. An Instrument Chunk defines basic parameters that an instrument (such as a sampling keyboard) can use to play back the sound data.

`MIDIDataID` The MIDI Data Chunk. A MIDI Chunk contains MIDI data.

`AudioRecordingID` The Audio Recording Chunk. An Audio Recording Chunk contains information pertaining to audio recording devices.

`ApplicationSpecificID` The Application Chunk. An Application Chunk contains application-specific information.

`CommentID`	The Comment Chunk. A Comment Chunk contains a comment.
`NameID`	The Name Chunk. A Name Chunk contains the name of the sampled sound.
`AuthorID`	The Author Chunk. An Author Chunk contains one or more names of the authors (or creators) of the sampled sound.
`CopyrightID`	The Copyright Chunk. A Copyright Chunk contains a copyright notice for the sampled sound.
`AnnotationID`	The Annotation Chunk. An Annotation Chunk contains a comment.

Data Structures

This section describes the data structures that the Sound Manager defines. The Sound Manager uses many of these data structures (such as sound headers) to store information about sounds or sound channels. You should use these data structures only if you need to access this information or to customize sound play. The Sound Manager also defines several data structures that allow you to control sound output or to receive information about its status.

You use the sound command record to define a sound command that you send to the Sound Manager using either the `SndDoCommand` or `SndDoImmediate` functions.

If you want to play only a portion of a sound, you can use an audio selection record in conjunction with the `SndStartFilePlay` function.

You use the sound channel status record to obtain information from the Sound Manager about a specific sound channel, and you use the Sound Manager status record to obtain information about all sound channels.

The sound channel record stores information about a sound channel. Many of the fields of this record are for internal Sound Manager use only, but there are a few that you can access directly.

The sound header record stores information about sampled-sound data. You can use a sound header record to obtain information on a sound or to change a sound's loop points. The extended sound header record and the compressed sound header record add several fields to the sound header record that provide more information about a sound.

If your application uses the `SndPlayDoubleBuffer` function to customize the double buffering of sound data, you need to set up a sound double buffer header record, which must include pointers to two sound double buffer records.

Sound Command Records

A **sound command record** describes a sound command that you send to a sound channel using the `SndDoCommand` or `SndDoImmediate` function. The `SndCommand` data type defines a sound command record.

```
TYPE SndCommand =
PACKED RECORD
   cmd:        Integer;    {command number}
   param1:     Integer;    {first parameter}
   param2:     LongInt;    {second parameter}
END;
```

Field descriptions

`cmd`	The number of the sound command you wish to execute.
`param1`	The first parameter of the sound command.
`param2`	The second parameter of the sound command.

The meaning of the `param1` and `param2` fields depends on the particular sound command being issued. See "Sound Command Numbers" beginning on page 2-92 for a description of the sound commands your application can use.

Audio Selection Records

You can pass a pointer to an audio selection record to the `SndStartFilePlay` function to play only part of a sound in a file on disk. The `AudioSelection` data type defines an audio selection record.

```
TYPE AudioSelection =
PACKED RECORD
   unitType:   LongInt;    {type of time unit}
   selStart:   Fixed;      {starting point of selection}
   selEnd:     Fixed;      {ending point of selection}
END;
```

Field descriptions

`unitType`	The type of unit of time used in the `selStart` and `selEnd` fields. You can set this to seconds by specifying the constant `unitTypeSeconds`.
`selStart`	The starting point in seconds of the sound to play. If `selStart` is greater than `selEnd`, `SndStartFilePlay` returns an error.
`selEnd`	The ending point in seconds of the sound to play.

Use a constant to specify the unit type.

```
CONST
   unitTypeSeconds         = $0000;    {seconds}
   unitTypeNoSelection     = $FFFF;    {no selection}
```

If the value in the `unitType` field is `unitTypeNoSelection`, then the values in the `selStart` and `selEnd` fields are ignored and the entire sound plays. Alternatively, if you wish to play an entire sound, you can pass `NIL` instead of a pointer to an audio selection record to the `SndStartFilePlay` function.

Sound Channel Status Records

To obtain information about a sound channel, you can pass a pointer to a **sound channel status record** to the `SndChannelStatus` function. The `SCStatus` data type defines a sound channel status record.

```
TYPE SCStatus =
RECORD
   scStartTime:            Fixed;      {starting time for play from disk}
   scEndTime:              Fixed;      {ending time for play from disk}
   scCurrentTime:          Fixed;      {current time for play from disk}
   scChannelBusy:          Boolean;    {TRUE if channel is processing cmds}
   scChannelDisposed:      Boolean;    {reserved}
   scChannelPaused:        Boolean;    {TRUE if channel is paused}
   scUnused:               Boolean;    {unused}
   scChannelAttributes:    LongInt;    {attributes of this channel}
   scCPULoad:              LongInt;    {CPU load for this channel}
END;
```

Field descriptions

`scStartTime` If the Sound Manager is playing from disk through the specified sound channel, then `scStartTime` is the starting time in seconds from the beginning of the sound for the play from disk. Otherwise, `scStartTime` is 0.

`scEndTime` If the Sound Manager is playing from disk through the specified sound channel, then `scEndTime` is the ending time in seconds from the beginning of the sound for the play from disk. Otherwise, `scEndTime` is 0.

`scCurrentTime` If the Sound Manager is playing from disk through the specified sound channel, then `scCurrentTime` is the current time in seconds from the beginning of the disk play. Otherwise, `scCurrentTime` is 0. The Sound Manager updates the value of this field only periodically, and you should not rely on the accuracy of its value.

`scChannelBusy` If the specified channel is currently processing sound commands, then `scChannelBusy` is `TRUE`; otherwise, `scChannelBusy` is `FALSE`.

`scChannelDisposed`
Reserved for use by Apple Computer, Inc.

`scChannelPaused`
If the Sound Manager is playing from disk through the specified sound channel and the play from disk is paused, then `scChannelPaused` is `TRUE`; otherwise, `scChannelPaused` is `FALSE`. This field is also `TRUE` if the channel was paused with the `pauseCmd` sound command.

`scUnused` Reserved for use by Apple Computer, Inc.

`scChannelAttributes`
: The current attributes of the specified channel. These attributes are in the channel initialization parameters format. The value returned in this field is always identical to the value passed in the `init` parameter to `SndNewChannel`.

`scCPULoad`
: The CPU load for the specified channel. You should not rely on the value in this field.

You can mask out certain values in the `scChannelAttributes` field to determine how a channel has been initialized.

```
CONST
   initPanMask    = $0003;    {mask for right/left pan values}
   initSRateMask  = $0030;    {mask for sample rate values}
   initStereoMask = $00C0;    {mask for mono/stereo values}
   initCompMask   = $FF00;    {mask for compression IDs}
```

Sound Manager Status Records

You can use the `SndManagerStatus` function to get a **Sound Manager status record,** which gives information on the current CPU loading caused by all open channels of sound. The `SMStatus` data type defines a Sound Manager status record.

```
TYPE SMStatus =
PACKED RECORD
   smMaxCPULoad:     Integer;    {maximum load on all channels}
   smNumChannels:    Integer;    {number of allocated channels}
   smCurCPULoad:     Integer;    {current load on all channels}
END;
```

Field descriptions

`smMaxCPULoad`
: The maximum CPU load that the Sound Manager will not exceed when allocating channels. The `smMaxCPULoad` field is set to a default value of 100 when the system starts up.

`smNumChannels`
: The number of sound channels that are currently allocated by all applications. This does not mean that the channels allocated are being used, only that they have been allocated and that CPU loading is being reserved for these channels.

`smCurCPULoad`
: The CPU load that is being taken up by currently allocated channels.

IMPORTANT

Although you can use the information contained in the Sound Manager status record to determine how many channels are allocated, you should not rely on the information in the `smMaxCPULoad` or `smCurCPULoad` field. To determine whether the Sound Manager can create a new channel, simply call the `SndNewChannel` function, which returns an appropriate result code if it is unable to allocate a new channel. ▲

Sound Channel Records

The Sound Manager maintains a sound channel record to store information about each sound channel that you allocate directly by calling the `SndNewChannel` function or indirectly by passing a `NIL` channel to a high-level Sound Manager routine like the `SndPlay` function. The `SndChannel` data type defines a sound channel record.

```
TYPE SndChannel =
PACKED RECORD
   nextChan:      SndChannelPtr; {pointer to next channel}
   firstMod:      Ptr;           {used internally}
   callBack:      ProcPtr;       {pointer to callback procedure}
   userInfo:      LongInt;       {free for application's use}
   wait:          LongInt;       {used internally}
   cmdInProgress: SndCommand;    {used internally}
   flags:         Integer;       {used internally}
   qLength:       Integer;       {used internally}
   qHead:         Integer;       {used internally}
   qTail:         Integer;       {used internally}
   queue:         ARRAY[0..stdQLength-1] OF SndCommand;
END;
```

Field descriptions

`nextChan` A pointer to the next sound channel in a single queue of channels that the Sound Manager maintains for all applications.

`firstMod` Used internally.

`callBack` A pointer to the callback procedure associated with the sound channel. See page 2-152 for information on this callback procedure.

`userInfo` A value that your application can use to store information.

`wait` Used internally.

`cmdInProgress` Used internally.

`flags` Used internally.

`qLength` Used internally.

`qHead` Used internally.

`qTail` Used internally.

`queue` The sound commands pending for the sound channel.

The only field of the sound channel record that you are likely to need to access directly is the `userInfo` field. This field is useful if you need to pass a value to a Sound Manager callback procedure or completion routine. For example, you might pass the value stored in the A5 register so that your callback procedure can access your application's global variables. Or, you might store a handle to sound data here so that a routine that disposes of an allocated channel can also release the sound data that the channel played.

In rarer instances, you might need to access the `callBack` field of the sound channel record directly. Ordinarily, you set this field by specifying a callback procedure when

you call the `SndNewChannel` function. However, you can change the callback procedure associated with a channel by changing this field directly. The Sound Manager will then execute the procedure you specify in this field whenever the channel processes a `callBackCmd` command.

▲ **WARNING**
You should not attempt to manipulate all open sound channels by using the `nextChan` field to walk the sound channel queue. The queue might contain channels opened by other applications. If you need to perform some operation on all sound channels that your application has allocated, you should maintain your own data structure that keeps track of your application's channels. ▲

Sound Header Records

Sound resources often contain sampled-sound data as well as sound commands. The sound data is contained in the last field of the sound header. You can access a sound header record to find information about sampled-sound data. The standard sound header is used only for simple monophonic sounds. The `SoundHeader` data type defines a sampled sound header record.

```
TYPE SoundHeader =
PACKED RECORD
   samplePtr:     Ptr;        {if NIL, samples in sampleArea}
   length:        LongInt;    {number of samples in array}
   sampleRate:    Fixed;      {sample rate}
   loopStart:     LongInt;    {loop point beginning}
   loopEnd:       LongInt;    {loop point ending}
   encode:        Byte;       {sample's encoding option}
   baseFrequency: Byte;       {base frequency of sample}
   sampleArea:    PACKED ARRAY[0..0] OF Byte;
END;
```

Field descriptions

`samplePtr` A pointer to the sampled-sound data. If the sampled sound is located in memory immediately after the `baseFrequency` field, then this field should be set to `NIL`. Otherwise, this field is a pointer to the memory location of the sampled-sound data. (This might be useful if you want to change some fields of a sound header but do not want to modify a handle to a sound resource directly.)

`length` The number of bytes of sound data.

`sampleRate` The rate at which the sample was originally recorded. The Sound Manager can play sounds sampled at any rate up to 64 kHz. The values corresponding to the three most common sample rates (11 kHz, 22 kHz, and 44 kHz) are defined by constants:

```
CONST
   rate44khz   = $AC440000;   {44100.00000 Fixed}
   rate22khz   = $56EE8BA3;   {22254.54545 Fixed}
   rate11khz   = $2B7745D1;   {11127.27273 Fixed}
```

Note that the sample rate is declared as a `Fixed` data type, but the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.

`loopStart` The starting point of the portion of the sampled sound header that is to be used by the Sound Manager when determining the duration of `freqDurationCmd`. These loop points specify the byte numbers in the sampled data to be used as the beginning and end points to cycle through when playing the sound. The loop starting and ending points are 0-based.

`loopEnd` The end point of the portion of the sampled sound header that is to be used by the Sound Manager when determining the duration of `freqDurationCmd`. If no looping is desired, set both `loopStart` and `loopEnd` to 0.

`encode` The method of encoding used to generate the sampled-sound data. The current encoding option values are

```
CONST
   stdSH    = $00;   {standard sound header}
   extSH    = $FF;   {extended sound header}
   cmpSH    = $FE;   {compressed sound header}
```

For a standard sound header, you should specify the constant `stdSH`. Encode option values in the ranges 0 through 63 and 128 to 255 are reserved for use by Apple. You are free to use numbers in the range 64 through 127 for your own encode options.

`baseFrequency` The pitch at which the original sample was taken. This value must be in the range 1 through 127. Table 2-2 on page 2-43 lists the possible `baseFrequency` values. The `baseFrequency` value allows the Sound Manager to calculate the proper playback rate of the sample when an application uses the `freqDurationCmd` command. Applications should not alter the `baseFrequency` field of a sampled sound; to play the sample at different pitches, use `freqDurationCmd` or `freqCmd`.

`sampleArea` If the value of `samplePtr` is `NIL`, this field is an array of bytes, each of which contains a value similar to the values in a wave-table description. These values are interpreted as offset values, where $80 represents an amplitude of 0. The value $00 is the most negative amplitude, and $FF is the largest positive amplitude. The samples are numbered 1 through the value in the `length` parameter.

If you need to create a sound header for sampled-sound data that your application has recorded, then you should use the `SetupSndHeader` function, described in the chapter "Sound Input Manager" in this book.

Extended Sound Header Records

For sampled-sound data that is more complex than a standard sound header can describe, the Sound Manager uses an extended sound header record. Sound data described by such a header can be monophonic or stereo, but it cannot be compressed.

Most of the fields of the extended sound header correspond to fields of the sampled sound header. However, the extended sound header allows the encoding of stereo sound. The `numChannels` field contains the number of channels of sound recorded, and the `numFrames` field contains the number of frames of sound recorded in each channel. For more information on the format of sampled sound frames, see "Sound Files" on page 2-81.

Note

The word "channel" can be confusing in this context, because a sound resource containing polyphonic sound (that is, multichannel sound) can be played on a single Sound Manager sound channel. **Channel** is a general term for the portion of sound data that can be described by a single sound wave. Monophonic sound is composed of a single channel. **Stereo sound** (also called **polyphonic sound**) is composed of several channels of sound played simultaneously. "Sound channel" is a term specific to the Sound Manager. ◆

```
TYPE ExtSoundHeader =
PACKED RECORD
   samplePtr:        Ptr;        {if NIL, samples in sampleArea}
   numChannels:      LongInt;    {number of channels in sample}
   sampleRate:       Fixed;      {rate of original sample}
   loopStart:        LongInt;    {loop point beginning}
   loopEnd:          LongInt;    {loop point ending}
   encode:           Byte;       {sample's encoding option}
   baseFrequency:    Byte;       {base freq. of original sample}
   numFrames:        LongInt;    {total number of frames}
   AIFFSampleRate:   Extended80; {rate of original sample}
   markerChunk:      Ptr;        {reserved}
   instrumentChunks: Ptr;        {pointer to instrument info}
   AESRecording:     Ptr;        {pointer to audio info}
   sampleSize:       Integer;    {number of bits per sample}
   futureUse1:       Integer;    {reserved}
   futureUse2:       LongInt;    {reserved}
   futureUse3:       LongInt;    {reserved}
   futureUse4:       LongInt;    {reserved}
   sampleArea:       PACKED ARRAY[0..0] OF Byte;
END;
```

Field descriptions

`samplePtr`	A pointer to the sampled-sound data. If the sampled sound is located in memory immediately after the `futureUse4` field, then this field should be set to `NIL`. Otherwise, this field is a pointer to the memory location of the sampled-sound data.
`numChannels`	The number of channels in the sampled-sound data.
`sampleRate`	The rate at which the sample was originally recorded. The approximate sample rates are shown in Table 2-1 on page 2-16. Note that the sample rate is declared as a `Fixed` data type, but the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.
`loopStart`	The starting point of the portion of the extended sampled sound header that is to be used by the Sound Manager when determining the duration of `freqDurationCmd`. These loop points specify the byte numbers in the sampled data to be used as the beginning and end points to cycle through when playing the sound. The loop starting and ending points are 0-based.
`loopEnd`	The end point of the portion of the extended sampled sound header that is to be used by the Sound Manager when determining the duration of `freqDurationCmd`.
`encode`	The method of encoding used to generate the sampled-sound data. For an extended sound header, you should specify the constant `extSH`. Encode option values in the ranges 0 through 63 and 128 to 255 are reserved for use by Apple. You are free to use numbers in the range 64 through 127 for your own encode options.
`baseFrequency`	The pitch at which the original sample was taken. This value must be in the range 1 through 127. Table 2-2 on page 2-43 lists the possible `baseFrequency` values. The `baseFrequency` value allows the Sound Manager to calculate the proper playback rate of the sample when an application uses the `freqDurationCmd` command. Applications should not alter the `baseFrequency` field of a sampled sound; to play the sample at different pitches, use `freqDurationCmd` or `freqCmd`.
`numFrames`	The number of frames in the sampled-sound data. Each frame contains `numChannels` bytes for 8-bit sound data.
`AIFFSampleRate`	The sample rate at which the frames were sampled before compression, as expressed in the 80-bit extended data type representation.
`markerChunk`	Synchronization information. The `markerChunk` field is not presently used and should be set to `NIL`.
`instrumentChunks`	Instrument information.
`AESRecording`	Information related to audio recording devices.
`sampleSize`	The number of bits in each sample frame.
`futureUse1`	Reserved.
`futureUse2`	Reserved.
`futureUse3`	Reserved.

`futureUse4` The four `futureUse` fields are reserved for use by Apple. To maintain compatibility with future releases of system software, you should always set these fields to 0.

`sampleArea` An array of interleaved sample points, each of which contains a value similar to the values in a wave-table description. For 8-bit sampled-sound data, these values are interpreted as offset values, where $80 represents an amplitude of 0. The value $00 is the largest negative amplitude, and $FF is the largest positive amplitude.

To compute the total number of bytes of a sample, multiply the values in the `numChannels`, `numFrames`, and `sampleSize` fields and divide by the number of bytes per sample (typically 8 or 16).

Note

Although extended sound headers (and compressed sound headers, described next) support the storage of 16-bit sound, only versions 3.0 and later of the Sound Manager can play 16-bit sounds. If your application uses 16-bit sound, you must convert it to 8-bit sound before earlier versions of the Sound Manager can play it. ◆

Compressed Sound Header Records

To describe compressed sampled-sound data, the Sound Manager uses a compressed sound header record. Compressed sound headers include all of the essential fields of extended sound headers in addition to several fields that pertain to compression. The `CmpSoundHeader` data type defines the compressed sound header record.

```
TYPE CmpSoundHeader =
PACKED RECORD
   samplePtr:        Ptr;          {if NIL, samples in sampleArea}
   numChannels:      LongInt;      {number of channels in sample}
   sampleRate:       Fixed;        {rate of original sample}
   loopStart:        LongInt;      {loop point beginning}
   loopEnd:          LongInt;      {loop point ending}
   encode:           Byte;         {sample's encoding option}
   baseFrequency:    Byte;         {base freq. of original sample}
   numFrames:        LongInt;      {length of sample in frames}
   AIFFSampleRate:   Extended80;   {rate of original sample}
   markerChunk:      Ptr;          {reserved}
   format:           OSType;       {data format type}
   futureUse2:       LongInt;      {reserved}
   stateVars:        StateBlockPtr; {pointer to StateBlock}
   leftOverSamples:  LeftOverBlockPtr;
                                   {pointer to LeftOverBlock}
   compressionID:    Integer;      {ID of compression algorithm}
   packetSize:       Integer;      {number of bits per packet}
   snthID:           Integer;      {unused}
```

```
   sampleSize:       Integer;       {bits in each sample point}
   sampleArea:       PACKED ARRAY[0..0] OF Byte;
END;
```

Field descriptions

`samplePtr`	The location of the compressed sound frames. If `samplePtr` is `NIL`, then the frames are located in the `sampleArea` field of the compressed sound header. Otherwise, `samplePtr` points to a buffer that contains the frames.
`numChannels`	The number of channels in the sample.
`sampleRate`	The sample rate at which the frames were sampled before compression. The approximate sample rates are shown in Table 2-1 on page 2-16. Note that the sample rate is declared as a `Fixed` data type, but the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.
`loopStart`	The beginning of the loop points of the sound before compression. The loop starting and ending points are 0-based.
`loopEnd`	The end of the loop points of the sound before compression.
`encode`	The method of encoding (if any) used to generate the sampled-sound data. For a compressed sound header, you should specify the constant `cmpSH`. Encode option values in the ranges 0 through 63 and 128 to 255 are reserved for use by Apple. You are free to use numbers in the range 64 through 127 for your own encode options.
`baseFrequency`	The pitch of the original sampled sound. It is not used by `bufferCmd`. If you wish to make use of `baseFrequency` with a compressed sound, you must first expand it and then play it with `soundCmd` and `freqDurationCmd`.
`numFrames`	The number of frames contained in the compressed sound header. When you store multiple channels of noncompressed sound, store them as interleaved sample frames (as in AIFF). When you store multiple channels of compressed sounds, store them as interleaved packet frames.
`AIFFSampleRate`	The sample rate at which the frames were sampled before compression, as expressed in the 80-bit extended data type representation.
`markerChunk`	Synchronization information. The `markerChunk` field is not presently used and should be set to `NIL`.
`format`	The data format type. This field contains a value of type `OSType` that defines the compression algorithm, if any, used to generate the audio data. For example, for data generated using MACE 3:1 compression, this field should contain the value `'MAC3'`. See page 2-86 for a list of the format types defined by Apple. This field is used only if the `compressionID` field contains the value `fixedCompression`.

`futureUse2` This field is reserved for use by Apple. To maintain compatibility with future releases of system software, you should always set this field to 0.

`stateVars` A pointer to a state block. This field is used to store the state variables for a given algorithm across consecutive calls. See "State Blocks" on page 2-119 for a description of the state block.

`leftOverSamples` A pointer to a leftover block. You can use this block to store samples that will be truncated across algorithm invocations. See "Leftover Blocks" on page 2-119 for a description of the leftover block.

`compressionID` The compression algorithm used on the samples in the compressed sound header. You can use a constant to define the compression algorithm.

```
CONST
   variableCompression
                     = -2; {variable-ratio compr.}
   fixedCompression  = -1; {fixed-ratio compr.}
   notCompressed     = 0;  {noncompressed samples}
   threeToOne        = 3;  {3:1 compressed samples}
   sixToOne          = 4;  {6:1 compressed samples}
```

The constant `fixedCompression` is available only with Sound Manager versions 3.0 and later. If the `compressionID` field contains the value `fixedCompression`, the Sound Manager reads the `format` field to determine the compression algorithm used to generate the compressed data. Otherwise, the Sound Manager reads the `compressionID` field. Apple reserves the right to use compression IDs in the range 0 through 511. Currently the constant `variableCompression` is not used by the Sound Manager.

`packetSize` The size, in bits, of the smallest element that a given expansion algorithm can work with. You can use a constant to define the packet size.

```
CONST
   sixToOnePacketSize   = 8;  {size for 6:1}
   threeToOnePacketSize = 16; {size for 3:1}
```

Beginning with Sound Manager version 3.0, you can specify the value 0 in this field to instruct the Sound Manager to determine the packet size itself.

`snthID` This field is unused. You should set it to 0.

`sampleSize` The size of the sample before it was compressed. The samples passed in the compressed sound header should always be byte-aligned, and any padding done to achieve byte alignment should be done from the left with zeros.

`sampleArea` The sample frames, but only when the `samplePtr` field is `NIL`. Otherwise, the sample frames are in the location indicated by `samplePtr`.

Sound Double Buffer Header Records

You must fill in a **sound double buffer header record** and two sound double buffer records if you wish to manage your own double buffers. The `SndDoubleBufferHeader` data type defines a sound double buffer header.

```
TYPE SndDoubleBufferHeader =
PACKED RECORD
   dbhNumChannels:      Integer;       {number of sound channels}
   dbhSampleSize:       Integer;       {sample size, if noncompressed}
   dbhCompressionID:    Integer;       {ID of compression algorithm}
   dbhPacketSize:       Integer;       {number of bits per packet}
   dbhSampleRate:       Fixed;         {sample rate}
   dbhBufferPtr:        ARRAY[0..1] OF SndDoubleBufferPtr;
                                       {pointers to SndDoubleBuffer}
   dbhDoubleBack:       ProcPtr;       {pointer to doubleback procedure}
END;
```

Sound Manager versions 3.0 and later support custom compression and decompression algorithms by defining the revised sound double buffer header record, of type `SndDoubleBufferHeader2`. It's identical to the `SndDoubleBufferHeader` data type except that it contains the `dbhFormat` field at the end.

```
TYPE SndDoubleBufferHeader2 =
PACKED RECORD
   dbhNumChannels:      Integer;       {number of sound channels}
   dbhSampleSize:       Integer;       {sample size, if noncompressed}
   dbhCompressionID:    Integer;       {ID of compression algorithm}
   dbhPacketSize:       Integer;       {number of bits per packet}
   dbhSampleRate:       Fixed;         {sample rate}
   dbhBufferPtr:        ARRAY[0..1] OF SndDoubleBufferPtr;
                                       {pointers to SndDoubleBuffer}
   dbhDoubleBack:       ProcPtr;       {pointer to doubleback procedure}
   dbhFormat:           OSType;        {signature of codec}
END;
```

Field descriptions

`dbhNumChannels`
The number of channels for the sound (1 for monophonic sound, 2 for stereo).

`dbhSampleSize` The sample size for the sound if the sound is not compressed. If the sound is compressed, `dbhSampleSize` should be set to 0. Samples

that are 1–8 bits have a `dbhSampleSize` value of 8; samples that are 9–16 bits have a `dbhSampleSize` value of 16. Currently, only 8-bit samples are supported. For further information on sample sizes, refer to the AIFF specification.

`dbhCompressionID`
The compression identification number of the compression algorithm, if the sound is compressed. If the sound is not compressed, `dbhCompressionID` should be set to 0.

`dbhPacketSize` The packet size in bits for the compression algorithm specified by `dbhCompressionID`, if the sound is compressed.

`dbhSampleRate` The sample rate for the sound. Note that the sample rate is declared as a `Fixed` data type, but the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.

`dbhBufferPtr` An array of two pointers, each of which should point to a valid `SndDoubleBuffer` record.

`dbhDoubleBack` A pointer to the application-defined routine that is called when the double buffers are switched and the exhausted buffer needs to be refilled.

`dbhFormat` The data format type. This field contains a value of type `OSType` that defines the compression algorithm, if any, to be used to decompress the audio data. For example, for data generated using MACE 3:1 compression, this field should contain the value `'MAC3'`. See page 2-86 for a list of the format types defined by Apple. This field is used only if the `dbhCompressionID` field contains the value `fixedCompression`.

The `dbhBufferPtr` array contains pointers to two sound double buffer records, whose format is defined below. These are the two buffers between which the Sound Manager switches until all the sound data has been sent into the sound channel. When you make the call to `SndPlayDoubleBuffer`, the two buffers should both already contain a nonzero number of frames of data.

Sound Double Buffer Records

You must fill in a **sound double buffer header record** if you wish to manage your own double buffers. The `dbhBufferPtr` field of the sound double buffer header record references two sound double buffer records, which you must also fill out. The `SndDoubleBufferHeader` data type defines a sound double buffer header.

```
TYPE SndDoubleBuffer =
PACKED RECORD
   dbNumFrames:   LongInt;                    {number of frames in buffer}
   dbFlags:       LongInt;                    {buffer status flags}
   dbUserInfo:    ARRAY[0..1] OF LongInt;     {for application's use}
   dbSoundData:   PACKED ARRAY[0..0] OF Byte; {array of data}
END;
```

Field descriptions

`dbNumFrames`	The number of frames in the `dbSoundData` array.
`dbFlags`	Buffer status flags.
`dbUserInfo`	Two long words into which you can place information that you need to access in your doubleback procedure.
`dbSoundData`	A variable-length array. You write samples into this array, and the Sound Manager reads samples out of this array.

The buffer status flags field for each of the two buffers can contain either of these values that your doubleback procedure must set when appropriate:

```
CONST
   dbBufferReady     = $00000001;
   dbLastBuffer      = $00000004;
```

All other bits in the `dbFlags` field are reserved by Apple; your application should not modify them.

Chunk Headers

Every chunk in an AIFF or AIFF-C file contains a **chunk header** that defines characteristics of the chunk. The `ChunkHeader` data type defines a chunk header.

```
TYPE ChunkHeader =
RECORD
   ckID:    ID;         {chunk type ID}
   ckSize:  LongInt;    {number of bytes of data}
END;
```

Field descriptions

`ckID`	The ID of the chunk. An ID is a 32-bit concatenation of any four printable ASCII characters in the range `' '` (space character, ASCII value $20) through `'~'` (ASCII value $7E). Spaces cannot precede printing characters, but trailing spaces are allowed. Control characters are not allowed. See "Chunk IDs" on page 2-98 for a list of the currently recognized chunk IDs.
`ckSize`	The size of the chunk in bytes, not including the `ckID` and `ckSize` fields.

Form Chunks

All sound files begin with a Form Chunk. This chunk defines the type and size of the file and can be thought of as enclosing the remaining chunks in the sound file. The `ContainerChunk` data type defines a Form Chunk.

```
TYPE ContainerChunk =
RECORD
   ckID:       ID;         {'FORM'}
   ckSize:     LongInt;    {number of bytes of data}
   formType:   ID;         {type of file}
END;
```

Field descriptions

`ckID` The ID of this chunk. For a Form Chunk, this ID is `'FORM'`.

`ckSize` The size of the data portion of this chunk. Note that the data portion of a Form Chunk is divided into two parts, `formType` and the remaining chunks of the sound file.

`formType` The type of audio file. For AIFF files, `formType` is `'AIFF'`. For AIFF-C files, `formType` is `'AIFC'`.

The size of an entire sound file is `ckSize+8`, because the `ckSize` field incorporates the size of all chunks of the sound file, except the sizes of the `ckID` and `ckSize` fields of the Form Chunk itself.

Format Version Chunks

AIFF-C files each contain exactly one Format Version Chunk, but files of type AIFF do not contain any. You can examine the Format Version Chunk to ensure that your application can process an AIFF-C file. The `FormatVersionChunk` data type defines a Format Version Chunk.

```
TYPE FormatVersionChunk =
RECORD
   ckID:       ID;         {'FVER'}
   ckSize:     LongInt;    {4}
   timestamp:  LongInt;    {date of format version}
END;
```

Field descriptions

`ckID` The ID of this chunk. For a Format Version Chunk, this ID is `'FVER'`.

`ckSize` The size of the data portion of this chunk. This value is always `4` in a Format Version Chunk because the `timestamp` field is 4 bytes long (the 8 bytes used by the `ckID` and `ckSize` fields are not included).

`timestamp` An indication of when the format version for this kind of file was created. The value indicates the number of seconds between midnight, January 1, 1904, and the time at which the AIFF-C file format was created.

Common Chunks

Every AIFF and AIFF-C file contains a Common Chunk that defines some fundamental characteristics of the sampled sound contained in the file. The format of the Common Chunk is different for AIFF and AIFF-C files. As a result, you need to determine the type of file format (by inspecting the `formType` field of the Form Chunk) before reading the Common Chunk.

For AIFF files, the `CommonChunk` data type defines a Common Chunk.

```
TYPE CommonChunk =
RECORD
   ckID:             ID;         {'COMM'}
   ckSize:           LongInt;    {size of chunk data}
   numChannels:      Integer;    {number of channels}
   numSampleFrames:  LongInt;    {number of sample frames}
   sampleSize:       Integer;    {number of bits per sample}
   sampleRate:       Extended;   {number of frames per second}
END;
```

Field descriptions

`ckID` The ID of this chunk. For a Common Chunk, this ID is `'COMM'`.

`ckSize` The size of the data portion of this chunk. In AIFF files, this field is always 18 because the 8 bytes used by the `ckID` and `ckSize` fields are not included.

`numChannels` The number of audio channels contained in the sampled sound. A value of 1 indicates monophonic sound, a value of 2 indicates stereo sound, a value of 4 indicates four-channel sound, and so forth.

`numSampleFrames` The number of sample frames in the Sound Data Chunk. Note that this field contains the number of sample frames, not the number of bytes of data and not the number of sample points. For noncompressed sound data, the total number of sample points in the file is `numChannels` * `numSampleFrames`.

`sampleSize` The number of bits in each sample point of noncompressed sound data. The `sampleSize` field can contain any integer from 1 to 32. For compressed sound data, this field indicates the number of bits per sample in the original sound data, before compression.

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

Extended Common Chunks

An AIFF-C file contains an extended Common Chunk that includes all of the fields of the Common Chunk, but adds two fields that describe the type of compression (if any) used on the audio data. The `ExtCommonChunk` data type defines an extended Common Chunk.

```
TYPE ExtCommonChunk =
RECORD
   ckID:             ID;         {'COMM'}
   ckSize:           LongInt;    {size of chunk data}
   numChannels:      Integer;    {number of channels}
   numSampleFrames:  LongInt;    {number of sample frames}
   sampleSize:       Integer;    {number of bits per sample}
   sampleRate:       Extended;   {number of frames per second}
   compressionType:  ID;         {compression type ID}
   compressionName:  PACKED ARRAY[0..0] OF Byte;
                                 {compression type name}
END;
```

Field descriptions

`ckID` The ID of this chunk. For an extended Common Chunk, this ID is `'COMM'`.

`ckSize` The size of the data portion of this chunk. For an extended Common Chunk, this size is 22 plus the number of bytes in the `compressionName` string.

`numChannels` The number of audio channels contained in the sampled sound. A value of 1 indicates monophonic sound, a value of 2 indicates stereo sound, a value of 4 indicates four-channel sound, and so forth.

`numSampleFrames` The number of sample frames in the Sound Data Chunk. Note that this field contains the number of sample frames, not the number of bytes of data and not the number of sample points. For noncompressed sound data, the total number of sample points in the file is `numChannels` * `numSampleFrames`.

`sampleSize` The number of bits in each sample point of noncompressed sound data. The `sampleSize` field can contain any integer from 1 to 32. For compressed sound data, this field indicates the number of bits per sample in the original sound data, before compression.

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

`compressionType` The ID of the compression algorithm, if any, used on the sound data. Compression algorithms supplied by Apple have the following types:

```
CONST
   NoneType                = 'NONE';
   ACE2Type                = 'ACE2';
   ACE8Type                = 'ACE8';
   MACE3Type               = 'MAC3';
   MACE6Type               = 'MAC6';
```

You can define your own compression types, but you should register them with Apple.

`compressionName`

A human-readable name for the compression algorithm ID specified in the `compressionType` field. If the number of bytes in this field is odd, then it is padded with the digit 0. Compression algorithms supplied by Apple have the following names:

```
CONST
   NoneName                = 'not compressed';
   ACE2to1Name             = 'ACE 2-to-1';
   ACE8to3Name             = 'ACE 8-to-3';
   MACE3to1Name            = 'MACE 3-to-1';
   MACE6to1Name            = 'MACE 6-to-1';
```

You can define your own compression types, but you should register them with Apple.

Sound Data Chunks

AIFF and AIFF-C files generally contain a Sound Data Chunk that contains the actual sampled-sound data. The `SoundDataChunk` data type defines a Sound Data Chunk.

```
TYPE SoundDataChunk =
RECORD
   ckID:       ID;         {'SSND'}
   ckSize:     LongInt;    {size of chunk data}
   offset:     LongInt;    {offset to sound data}
   blockSize:  LongInt;    {size of alignment blocks}
END;
```

Field descriptions

`ckID` The ID of this chunk. For a Sound Data Chunk, this ID is `'SSND'`.

`ckSize` The size of the data portion of this chunk. This size does not include the 8 bytes occupied by the values in the `ckID` and the `ckSize` fields.

`offset` An offset (in bytes) to the beginning of the first sample frame in the chunk data. Most applications do not need to use the offset field and should set it to 0.

`blockSize` The size (in bytes) of the blocks to which the sound data is aligned. This field is used in conjunction with the `offset` field for aligning sound data to blocks. As with the `offset` field, most applications do not need to use the `blockSize` field and should set it to 0.

The sampled-sound data follows the `blockSize` field. If the data following the `blockSize` field contains an odd number of bytes, a pad byte with a value of 0 is added at the end to preserve an even length for this chunk. If there is a pad byte, it is not

included in the ckSize field. For information on the format of the sampled-sound data, see "Sound Files" on page 2-81.

Version Records

The functions SndSoundManagerVersion and MACEVersion return version information using a **version record.** The NumVersion data type defines a version record.

```
TYPE NumVersion =
PACKED RECORD
CASE INTEGER OF
 0:
   (majorRev:        SignedByte;    {major revision level in BCD}
   minorAndBugRev:   SignedByte;    {minor revision level}
   stage:            SignedByte;    {development stage}
   nonRelRev:        SignedByte);   {nonreleased revision level}
 1:
   (version:         LongInt);      {all 4 fields together}
END;
```

IMPORTANT

A version record has the same structure as the first four fields of a version resource (a resource of type 'vers'). See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information about version resources. ▲

Field descriptions

majorRev — The major revision level. This field is a signed byte in binary-coded decimal format.

minorAndBugRev — The minor revision level. This field is a signed byte in binary-coded decimal format.

stage — The development stage. You should use the following constants to specify a development stage:

```
CONST
   developStage      = $20;   {prealpha release}
   alphaStage        = $40;   {alpha release}
   betaStage         = $60;   {beta release}
   finalStage        = $80;   {final release}
```

nonRelRev — The revision level of a prereleased version.

version — A long integer that contains all four version fields.

Leftover Blocks

The `leftOverSamples` field of a compressed sound header contains a pointer to a leftover block, defined by the `LeftOverBlock` data type.

```
TYPE LeftOverBlock =
RECORD
   count:         LongInt;
   sampleArea:    PACKED ARRAY[0..leftOverBlockSize - 1] OF Byte;
END;
```

Field descriptions

`count` The number of bytes in the `sampleArea` field.

`sampleArea` An array of bytes. This field contains samples that are truncated across invocations of the compression algorithm. The size of this field is defined by a constant.

```
CONST
   leftOverBlockSize       = 32;
```

State Blocks

The `stateVars` field of a compressed sound header contains a pointer to a state block, defined by the `StateBlock` data type.

```
TYPE StateBlock =
RECORD
   stateVar:         ARRAY[0..stateBlockSize - 1] OF Integer;
END;
```

Field descriptions

`stateVar` An array of integers. This field contains state variables that need to be preserved across invocations of the compression algorithm. The size of this field is defined by a constant.

```
CONST
   stateBlockSize          = 64;
```

Sound Manager Routines

This section describes the routines provided by the Sound Manager. You can use these routines to

- play sound resources
- play sounds stored in files directly from disk
- allocate and release sound channels

- send commands to a sound channel
- obtain information about the Sound Manager, a sound channel, all sound channels, or the system alert sound's status
- compress and expand audio data
- manage the reading and writing of double sound buffers

The section "Application-Defined Routines" on page 2-151 describes routines that your application might need to define, including callback procedures, completion routines, and doubleback procedures.

Assembly-Language Note

Most Sound Manager routines are accessed through the `_SoundDispatch` selector. However, the `SndAddModifier`, `SndControl`, `SndDisposeChannel`, `SndDoCommand`, `SndDoImmediate`, `SndNewChannel`, and `SndPlay` functions and the `SysBeep` procedure are accessed through their own trap macros. See "Summary of the Sound Manager," which begins on page 2-157, for a list of trap selector numbers. ◆

Playing Sound Resources

You can use the `SysBeep` procedure to play the system alert sound. Alert sounds are stored in the System file as format 1 `'snd '` resources. You can use the `SndPlay` function to play the sounds that are stored in any `'snd '` resource, either format 1 or format 2.

The `SysBeep` and `SndPlay` routines are the highest-level sound routines that the Sound Manager provides. Depending on the needs of your application, you might be able to accomplish all desired sound-related activity simply by using `SysBeep` to produce the system alert sound or by using `SndPlay` to play other sounds that are stored as `'snd '` resources.

SysBeep

You can use the `SysBeep` procedure to play the system alert sound.

```
PROCEDURE SysBeep (duration: Integer);
```

`duration` The duration (in ticks) of the resulting sound. This parameter is ignored except on a Macintosh Plus, Macintosh SE, or Macintosh Classic when the system alert sound is the Simple Beep. The recommended duration is 30 ticks, which equals one-half second.

DESCRIPTION

The `SysBeep` procedure causes the Sound Manager to play the system alert sound at its current volume. If necessary, the Sound Manager loads into memory the sound resource containing the system alert sound and links it to a sound channel. The user selects a system alert sound in the Alert Sounds subpanel of the Sound control panel.

The volume of the sound produced depends on the current setting of the system alert sound volume, which the user can adjust in the Alert Sounds subpanel of the Sound control panel. The system alert sound volume can also be read and set by calling the `GetSysBeepVolume` and `SetSysBeepVolume` routines. If the volume is set to 0 (silent) and the system alert sound is enabled, calling `SysBeep` causes the menu bar to blink once.

SPECIAL CONSIDERATIONS

Because the `SysBeep` procedure moves memory, you should not call it at interrupt time.

SEE ALSO

For information on enabling and disabling the system alert sound, see the description of `SndGetSysBeepState` and `SndGetSysBeepState` on page 2-137. For information on reading or adjusting the system alert sound volume, see "Controlling Volume Levels" beginning on page 2-139.

SndPlay

You can use the `SndPlay` function to play a sound resource that your application has loaded into memory.

```
FUNCTION SndPlay (chan: SndChannelPtr; sndHdl: Handle;
                  async: Boolean): OSErr;
```

`chan` A pointer to a valid sound channel. You can pass `NIL` instead of a pointer to a sound channel if you want the Sound Manager to internally allocate a sound channel in your application's heap zone.

`sndHdl` A handle to the sound resource to play.

`async` A Boolean value that indicates whether the sound should be played asynchronously (`TRUE`) or synchronously (`FALSE`). This parameter is ignored (and the sound plays synchronously) if `NIL` is passed in the first parameter.

DESCRIPTION

The `SndPlay` function attempts to play the sound located at `sndHdl`, which is expected to have the structure of a format 1 or format 2 `'snd '` resource. If the resource has not yet been loaded, the `SndPlay` function fails and returns the `resProblem` result code.

All commands and data contained in the sound handle are then sent to the channel. Note that you can pass `SndPlay` a handle to some data created by calling the Sound Input Manager's `SndRecord` function as well as a handle to an actual `'snd '` resource that you have loaded into memory.

▲ **WARNING**
In some versions of system software prior to system software version 7.0, the `SndPlay` function will not work properly with sound resources that specify the sound data type twice. This might happen if a resource specifies that a sound consists of sampled-sound data and an application does the same when creating a sound channel. For more information on this problem, see "Allocating Sound Channels" on page 2-20. ▲

The `chan` parameter is a pointer to a sound channel. If `chan` is not `NIL`, it is used as a valid channel. If `chan` is `NIL`, an internally allocated sound channel is used. If you do supply a sound channel pointer in the `chan` parameter, you can play the sound asynchronously. When a sound is played asynchronously, a callback procedure can be called when a `callBackCmd` command is processed by the channel. (This procedure is the callback procedure supplied to `SndNewChannel`.) See "Playing Sounds Asynchronously" on page 2-46 for more information on playing sounds asynchronously. The handle you pass in the `sndHdl` parameter must be locked for as long as the sound is playing asynchronously.

If a format 1 `'snd '` resource does not specify which type of sound data is to be played, `SndPlay` defaults to square-wave data. `SndPlay` also supports format 2 `'snd '` resources using sampled-sound data and a `bufferCmd` command. Note that to use `SndPlay` and sampled-sound data with a format 1 `'snd '` resource, the resource must include a `bufferCmd` command.

SPECIAL CONSIDERATIONS

Because the `SndPlay` function moves memory, you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`notEnoughHardwareErr`	–201	Insufficient hardware available
`resProblem`	–204	Problem loading the resource
`badChannel`	–205	Channel is corrupt or unusable
`badFormat`	–206	Resource is corrupt or unusable

SEE ALSO

For an example of how to play a sound resource using the `SndPlay` function, see the chapter "Introduction to Sound on the Macintosh" in this book.

For information on playing a sound resource without using the `SndPlay` function, see "Playing Sounds Using Low-Level Routines" on page 2-61.

Playing From Disk

Use the `SndStartFilePlay`, `SndPauseFilePlay`, and `SndStopFilePlay` functions to manage a continuous play from disk.

SndStartFilePlay

You can call the `SndStartFilePlay` function to initiate a play from disk.

```
FUNCTION SndStartFilePlay (chan: SndChannelPtr; fRefNum: Integer;
                           resNum: Integer; bufferSize: LongInt;
                           theBuffer: Ptr;
                           theSelection: AudioSelectionPtr;
                           theCompletion: ProcPtr;
                           async: Boolean): OSErr;
```

`chan` A pointer to a valid sound channel. You can pass `NIL` instead of a pointer to a sound channel if you want the Sound Manager to internally allocate a sound channel in your application's heap zone.

`fRefNum` The file reference number of the AIFF or AIFF-C file to play. To play a sound resource rather than a sound file, this field should be 0.

`resNum` The resource ID number of a sound resource to play. To play a sound file rather than a sound resource, this field should be 0.

`bufferSize`
The number of bytes of memory that the Sound Manager is to use for input buffering while reading in sound data. For `SndStartFilePlay` to execute successfully on the slowest Macintosh computers, use a buffer of at least 20,480 bytes. You can pass the value 0 to instruct the Sound Manager to allocate a buffer of the default size.

`theBuffer` A pointer to a buffer that the Sound Manager should use for input buffering while reading in sound data. If this parameter is `NIL`, the Sound Manager allocates two buffers, each half the size of the value specified in the `bufferSize` parameter. If this parameter is not `NIL`, the buffer should be a nonrelocatable block of size `bufferSize`.

`theSelection`
A pointer to an audio selection record that specifies which portion of a sound should be played. You can pass `NIL` to specify that the Sound Manager should play the entire sound.

`theCompletion`
A pointer to a completion routine that the Sound Manager calls when the sound is finished playing. You can pass `NIL` to specify that the Sound Manager should not execute a completion routine. This field is useful only for asynchronous play.

`async` A Boolean value that indicates whether the sound should be played asynchronously (`TRUE`) or synchronously (`FALSE`). You can play sound asynchronously only if you allocate your own sound channel (using `SndNewChannel`). If you pass `NIL` in the `chan` parameter and `TRUE` for this parameter, the `SndStartFilePlay` function returns the `badChannel` result code.

DESCRIPTION

The `SndStartFilePlay` function begins a continuous play from disk on a sound channel. The `chan` parameter is a pointer to the sound channel. If `chan` is not `NIL`, it is used as a valid channel. If `chan` is `NIL`, an internally allocated sound channel is used for play from disk. This internally allocated sound channel is not passed back to you. Because `SndPauseFilePlay` and `SndStopFilePlay` require a sound-channel pointer, you must allocate your own channel if you wish to use those routines.

The sounds you wish to play can be stored either in a file or in an `'snd '` resource. If you are playing a file, then `fRefNum` should be the file reference number of the file to be played and the parameter `resNum` should be set to 0. If you are playing an `'snd '` resource, then `fRefNum` should be set to 0 and `resNum` should be the resource ID number (not the file reference number) of the resource to play.

▲ **WARNING**
The `SndStartFilePlay` function might not play `'snd '` resources from disk correctly. In particular, the function will not execute correctly if any resource in the resource file containing the `'snd '` resource you wish to play has been changed through a call to the `WriteResource` procedure and you have not updated the resource file using the `UpdateResFile` procedure. To avoid this and other problems, you should use the `SndStartFilePlay` function to play only sound files. ▲

SPECIAL CONSIDERATIONS

Because the `SndStartFilePlay` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndStartFilePlay` function are

Trap macro	Selector
`_SoundDispatch`	$0D000008

RESULT CODES

`noErr`	0	No error
`notEnoughHardwareErr`	–201	Insufficient hardware available
`queueFull`	–203	No room in the queue
`badChannel`	–205	Channel is corrupt or unusable
`badFormat`	–206	Resource is corrupt or unusable
`notEnoughBufferSpace`	–207	Insufficient memory available
`badFileFormat`	–208	File is corrupt or unusable, or not AIFF or AIFF-C
`channelBusy`	–209	Channel is busy
`buffersTooSmall`	–210	Buffer is too small
`siInvalidCompression`	–223	Invalid compression type

SEE ALSO

For an example of how to play a sound file, see the chapter "Introduction to Sound on the Macintosh" in this book.

For information on the format of a completion routine, see "Completion Routines" on page 2-151.

SndPauseFilePlay

You can use the `SndPauseFilePlay` function to toggle the state of a play from disk in progress, just as you might use the pause button on an audiocassette tape player to temporarily pause and then resume play.

```
FUNCTION SndPauseFilePlay (chan: SndChannelPtr): OSErr;
```

`chan` A pointer to a valid sound channel currently processing a play from disk initiated by a call to the `SndStartFilePlay` function.

DESCRIPTION

The `SndPauseFilePlay` function suspends the play from disk on the channel specified by the `chan` parameter if that play from disk is not already paused; the function resumes play if the play from disk is already paused.

The `SndPauseFilePlay` function is used in conjunction with `SndStopFilePlay` to control play from disk on a sound channel. Note that this call can be made only if your application has already called `SndStartFilePlay` with a valid sound channel. You cannot use this function with a synchronous call to `SndStartFilePlay` because, in that case, program control does not return to the caller until after the sound has completely finished playing.

If the channel specified by the `chan` parameter is not being used for play from disk, then `SndPauseFilePlay` returns the result code `channelNotBusy`. If the channel is busy

and paused, then play from disk is resumed. If the channel is busy and the channel is not paused, then play from disk is suspended.

SPECIAL CONSIDERATIONS

You can call the `SndPauseFilePlay` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndPauseFilePlay` function are

Trap macro	Selector
`_SoundDispatch`	$02040008

RESULT CODES

`noErr`	0	No error
`queueFull`	–203	No room in the queue
`badChannel`	–205	Channel is corrupt or unusable
`channelNotBusy`	–211	Channel not currently used

SndStopFilePlay

You can use `SndStopFilePlay` to stop an asynchronous play from disk.

```
FUNCTION SndStopFilePlay (chan: SndChannelPtr;
                          quietNow: Boolean): OSErr;
```

`chan` A pointer to a valid sound channel currently processing a play from disk initiated by a call to the `SndStartFilePlay` function.

`quietNow` A Boolean value that indicates whether the play from disk should be stopped immediately (`TRUE`) or when it completes execution (`FALSE`).

DESCRIPTION

The `SndStopFilePlay` function either can stop an asynchronous play from disk immediately or can take control of the CPU until a play from disk finishes. The `SndStopFilePlay` function does not return until all asynchronous file I/O calls have completed and any internally allocated memory has been released. If `async` is `FALSE`, then `SndStopFilePlay` lets the sound complete normally and returns only after the sound has completed, all asynchronous file I/O calls have completed, and any internal allocated memory has been released.

For example, you might use the function to stop the playing of a sound file if the user selects an option that turns off sound output while the file is already playing. In that case, you would pass `TRUE` to `quietNow`. Alternatively, you might have started a sound

playing asynchronously so that you could perform other tasks while the sound plays. But you might then finish those other tasks and want to convert the play from disk into a synchronous play. By passing `FALSE` to `quietNow`, you effectively achieve that.

SPECIAL CONSIDERATIONS

Because the `SndStopFilePlay` function might move memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndStopFilePlay` function are

Trap macro	Selector
`_SoundDispatch`	$03080008

RESULT CODES

`noErr`	0	No error
`badChannel`	–205	Channel is corrupt or unusable

Allocating and Releasing Sound Channels

If you use a high-level Sound Manager routine to play sounds, you might be able to let the Sound Manager internally allocate a sound channel. However, to use low-level sound commands or to take full advantage of the Sound Manager's high-level routines, you must allocate your own sound channels. The `SndNewChannel` function allows your application to allocate a new sound channel, and the `SndDisposeChannel` function allows your application to dispose of it.

SndNewChannel

You can use the `SndNewChannel` function to allocate a new sound channel.

```
FUNCTION SndNewChannel (VAR chan: SndChannelPtr; synth: Integer;
                        init: LongInt; userRoutine: ProcPtr):
                        OSErr;
```

`chan` A pointer to a sound channel record. You can pass a pointer whose value is `NIL` to force the Sound Manager to allocate the sound channel record internally.

`synth` The sound data type you intend to play on this channel. If you do not want to specify a specific data type, pass 0 in this parameter. You might do this if you plan to use the channel to play a single sound resource that itself specifies the sound's data type.

`init` The desired initialization parameters for the channel. If you cannot determine what types of sounds you will be playing on the channel, pass 0 in this parameter. Only sounds defined by wave-table data and sampled-sound data currently use the `init` options. You can use the `Gestalt` function to determine if a sound feature (such as stereo output) is supported by a particular computer.

`userRoutine`
A pointer to a callback procedure that the Sound Manager executes whenever it receives a `callBackCmd` command. If you pass `NIL` as the `userRoutine` parameter, then any `callBackCmd` commands sent to this channel are ignored.

DESCRIPTION

The `SndNewChannel` function internally allocates memory to store a queue of sound commands. If you pass a pointer to `NIL` as the `chan` parameter, the function also allocates a sound channel record in your application's heap and returns a pointer to that record. If you do not pass a pointer to `NIL` as the `chan` parameter, then that parameter must contain a pointer to a sound channel record.

If you pass a pointer to `NIL` as the `chan` parameter, then the amount of memory the `SndNewChannel` function allocates to store the sound commands is enough to store 128 sound commands. However, if you pass a pointer to the sound channel record rather than a pointer to `NIL`, the amount of memory allocated is determined by the `qLength` field of the sound channel record. Thus, if you wish to control the size of the sound queue, you must allocate your own sound channel record. Regardless of whether you allocate your own sound channel record, the Sound Manager allocates memory for the sound command queue internally.

The `synth` parameter specifies the sound data type you intend to play on this channel. You can use these constants to specify the data type:

```
CONST
   squareWaveSynth         = 1;        {square-wave data}
   waveTableSynth          = 3;        {wave-table data}
   sampledSynth            = 5;        {sampled-sound data}
```

In Sound Manager versions earlier than version 3.0, only one data type can be produced at any one time. As a result, `SndNewChannel` may fail if you attempt to open a channel specifying a data type other than the one currently being played.

To specify a sound output device other than the current sound output device, pass the value `kUseOptionalOutputDevice` in the `synth` parameter and the signature of the desired sound output device component in the `init` parameter.

```
CONST
   kUseOptionalOutputDevice      = -1;
```

The ability to redirect output away from the current sound output device is intended for use by specialized applications that need to use a specific sound output device. In

general, your application should always send sound to the current sound output device selected by the user.

SPECIAL CONSIDERATIONS

Because the `SndNewChannel` function allocates memory, you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`resProblem`	–204	Problem loading the resource
`badChannel`	–205	Channel is corrupt or unusable

SEE ALSO

For an example of a routine that uses the `SndNewChannel` function, see Listing 2-1 on page 2-20.

For information on the format of a callback procedure, see "Callback Procedures" on page 2-152.

SndDisposeChannel

If you allocate a sound channel by calling the `SndNewChannel` function, you must release the memory it occupies by calling the `SndDisposeChannel` function.

```
FUNCTION SndDisposeChannel (chan: SndChannelPtr;
                            quietNow: Boolean): OSErr;
```

`chan` A pointer to a valid sound channel record.

`quietNow` A Boolean value that indicates whether the channel should be disposed immediately (`TRUE`) or after sound stops playing (`FALSE`).

DESCRIPTION

The `SndDisposeChannel` function disposes of the queue of sound commands associated with the sound channel specified in the `chan` parameter. If your application created its own sound channel record in memory or installed a sound as a voice in a channel, the Sound Manager does not dispose of that memory. The Sound Manager also does not release memory associated with a sound resource that you have played on a channel. You might use the `userInfo` field of the sound channel record to store the address of a sound handle you wish to release before disposing of the sound channel itself.

The SndDisposeChannel function can dispose of a channel immediately or wait until the queued commands are processed. If quietNow is set to TRUE, a flushCmd command and then a quietCmd command are sent to the channel bypassing the command queue. This removes all commands, stops any sound in progress, and closes the channel. If quietNow is set to FALSE, then the Sound Manager issues a quietCmd command only; it does not bypass the command queue, and it waits until the quietCmd command is processed before disposing of the channel.

SPECIAL CONSIDERATIONS

Because the SndDisposeChannel function might dispose of memory, you should not call it at interrupt time.

RESULT CODES

noErr	0	No error
badChannel	–205	Channel is corrupt or unusable

Sending Commands to a Sound Channel

Once a sound channel is opened, you can send commands to that channel by issuing requests with the SndDoCommand and SndDoImmediate functions.

The section "Sound Command Numbers" beginning on page 2-92 lists the sound commands that you can send using SndDoCommand, SndDoImmediate, or (in several cases) SndControl.

SndDoCommand

You can queue a command in a sound channel by calling the SndDoCommand function.

```
FUNCTION SndDoCommand (chan: SndChannelPtr; cmd: SndCommand;
                       noWait: Boolean): OSErr;
```

chan	A pointer to a valid sound channel.
cmd	A sound command to be sent to the channel specified in the chan parameter.
noWait	A flag indicating whether the Sound Manager should wait for a free space in a full queue (FALSE) or whether it should return immediately with a queueFull result code if the queue is full (TRUE).

DESCRIPTION

The SndDoCommand function sends the sound command specified in the cmd parameter to the end of the command queue of the channel specified in the chan parameter.

The noWait parameter has meaning only if a sound channel's queue of sound commands is full. If the noWait parameter is set to FALSE and the queue is full, the Sound Manager waits until there is space to add the command, thus preventing your application from doing other processing. If noWait is set to TRUE and the queue is full, the Sound Manager does not send the command and returns the queueFull result code.

SPECIAL CONSIDERATIONS

Whether SndDoCommand moves memory depends on the particular sound command you're sending it. Most of the available sound commands do not cause SndDoCommand to move memory and can therefore be issued at interrupt time. Moreover, you can sometimes safely send commands at interrupt time that would otherwise cause memory to move if you've previously issued the soundCmd sound command to preconfigure the channel at noninterrupt time.

RESULT CODES

noErr	0	No error
queueFull	–203	No room in the queue
badChannel	–205	Channel is corrupt or unusable

SEE ALSO

For an example of a routine that uses the SndDoCommand function, see Listing 2-15 on page 2-42.

SndDoImmediate

You can use the SndDoImmediate function to place a sound command in front of a sound channel's command queue.

```
FUNCTION SndDoImmediate (chan: SndChannelPtr; cmd: SndCommand):
                                OSErr;
```

chan	A pointer to a sound channel.
cmd	A sound command to be sent to the channel specified in the chan parameter.

DESCRIPTION

The SndDoImmediate function operates much like SndDoCommand, except that it bypasses the existing command queue of the sound channel and sends the specified command directly to the Sound Manager for immediate processing. This routine also overrides any waitCmd, pauseCmd, or syncCmd commands that might have already been processed. However, other commands already received by the Sound Manager will

not be interrupted by the `SndDoImmediate` function (although a `quietCmd` command sent via `SndDoImmediate` will quiet a sound already playing).

SPECIAL CONSIDERATIONS

Whether `SndDoImmediate` moves memory depends on the particular sound command you're sending it. Most of the available sound commands do not cause `SndDoImmediate` to move memory and can therefore be issued at interrupt time. Moreover, you can sometimes safely send commands at interrupt time that would otherwise cause memory to move if you've previously issued the `soundCmd` sound command to preconfigure the channel at noninterrupt time.

RESULT CODES

`noErr`	0	No error
`badChannel`	–205	Channel is corrupt or unusable

SEE ALSO

For an example of a routine that uses the `SndDoImmediate` function, see Listing 2-4 on page 2-26.

Obtaining Information

To obtain information about whether a computer supports certain sound features, you should use the `Gestalt` function, documented in *Inside Macintosh: Operating System Utilities*. Sometimes, however, you might need information the `Gestalt` function is not able to provide. The Sound Manager provides a number of routines that you can use to obtain additional sound-related information.

You can obtain the version numbers of the Sound Manager and the MACE tools by calling the `SndSoundManagerVersion` and `MACEVersion` functions, respectively. You can obtain information about a sound channel and about all sound channels by calling the `SndControl`, `SndChannelStatus`, and `SndManagerStatus` functions, respectively.

The Sound Manager includes two routines—`SndGetSysBeepState` and `SndSetSysBeepState`—that allow you to determine and alter the status of the system alert sound.

To play a sound resource using low-level Sound Manager routines, you need the address of the sound header stored in the sound resource. Sound Manager versions 3.0 and later provide the `GetSoundHeaderOffset` function that you can use to obtain that information.

SndSoundManagerVersion

You can use `SndSoundManagerVersion` to determine the version of the Sound Manager tools available on a computer.

```
FUNCTION SndSoundManagerVersion: NumVersion;
```

DESCRIPTION

The `SndSoundManagerVersion` function returns a version number that contains the same information as in the first 4 bytes of a `'vers'` resource. You might use the `SndSoundManagerVersion` function to determine if a computer has the enhanced Sound Manager, which is necessary for multichannel sound and for continuous plays from disk.

SPECIAL CONSIDERATIONS

You can call the `SndSoundManagerVersion` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndSoundManagerVersion` function are

Trap macro	Selector
`_SoundDispatch`	$000C0008

SEE ALSO

For information on how to use the `SndSoundManagerVersion` function to determine whether the enhanced Sound Manager is available, see "Obtaining Version Information" on page 2-34.

MACEVersion

You can use `MACEVersion` to determine the version of the MACE tools available on a machine.

```
FUNCTION MACEVersion: NumVersion;
```

DESCRIPTION

The `MACEVersion` function returns a version number that contains the same information as in the first 4 bytes of a `'vers'` resource.

SPECIAL CONSIDERATIONS

You can call the `MACEVersion` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `MACEVersion` function are

Trap macro	Selector
`_SoundDispatch`	$00000010

SndControl

You can obtain information about a sound data type by using the `SndControl` function. In Sound Manager version 3.0 and later, however, you virtually never need to call `SndControl`. The capabilities that `SndControl` provides are either provided by the `Gestalt` function or are no longer supported. The `SndControl` function is documented here for completeness only.

```
FUNCTION SndControl (id: Integer; VAR cmd: SndCommand): OSErr;
```

`id` The sound data type you want to get information about.

`cmd` A sound command.

DESCRIPTION

The `SndControl` function sends a control command directly to the Sound Manager to get information about a specific data type. The available data types are specified by constants:

```
CONST
   squareWaveSynth   = 1;     {square-wave data}
   waveTableSynth    = 3;     {wave-table data}
   sampledSynth      = 5;     {sampled-sound data}
```

You can call `SndControl` even if no channel has been created for the type of data you want to get information about. `SndControl` can be used with the `availableCmd` or `versionCmd` sound commands to request information. The requested information is returned in the sound command record specified by the `cmd` parameter.

IMPORTANT

The `SndControl` function can indicate only whether a particular data format supports some feature (for example, stereo output), not whether the available sound hardware also supports that feature. In general, you should use the `Gestalt` function to determine whether the sound features you need are available in the current operating environment. ▲

In Sound Manager version 2.0, you can also use the `totalLoadCmd` and `loadCmd` commands to get information about the amount of CPU time consumed by sound-related processing. However, these commands are not very accurate and are not supported by version 3.0 and later.

SPECIAL CONSIDERATIONS

You should not call the `SndControl` function at interrupt time.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See the list of sound commands in "Sound Command Numbers" beginning on page 2-92 for a complete description of the sound commands supported by `SndControl`.

SndChannelStatus

You can use the `SndChannelStatus` function to determine the status of a sound channel.

```
FUNCTION SndChannelStatus (chan: SndChannelPtr;
                           theLength: Integer;
                           theStatus: SCStatusPtr): OSErr;
```

`chan`	A pointer to a valid sound channel.
`theLength`	The size in bytes of the sound channel status record. You should set this field to `SizeOf(SCStatus)`.
`theStatus`	A pointer to a sound channel status record.

DESCRIPTION

If the `SndChannelStatus` function executes successfully, the fields of the record specified by `theStatus` accurately describe the sound channel specified by `chan`.

SPECIAL CONSIDERATIONS

You can call the `SndChannelStatus` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndChannelStatus` function are

Trap macro	Selector
`_SoundDispatch`	$00100008

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	A parameter is incorrect
`badChannel`	–205	Channel is corrupt or unusable

SEE ALSO

For information on the structure of a sound channel status record, see "Sound Channel Status Records" on page 2-101.

SndManagerStatus

You can use the `SndManagerStatus` function to determine information about all sound channels currently allocated.

```
FUNCTION SndManagerStatus (theLength: Integer;
                           theStatus: SMStatusPtr): OSErr;
```

`theLength` The size in bytes of the Sound Manager status record. You should set this field to `SizeOf(SMStatus)`.

`theStatus` A pointer to a Sound Manager status record.

DESCRIPTION

The `SndManagerStatus` function determines information about all currently allocated sound channels. If the `SndManagerStatus` function executes successfully, the fields of the record specified by `theStatus` accurately describe the current status of the Sound Manager.

SPECIAL CONSIDERATIONS

You can call the `SndManagerStatus` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndManagerStatus` function are

Trap macro	Selector
`_SoundDispatch`	$00140008

RESULT CODES

noErr	0	No error

SndGetSysBeepState

You can use the SndGetSysBeepState procedure to determine if the system alert sound is enabled.

```
PROCEDURE SndGetSysBeepState (VAR sysBeepState: Integer);
```

sysBeepState
On exit, the state of the system alert sound.

DESCRIPTION

The SndGetSysBeepState procedure returns one of two states in the sysBeepState parameter, either the sysBeepDisable or the sysBeepEnable constant.

```
CONST
   sysBeepDisable    = $0000;    {system alert sound disabled}
   sysBeepEnable     = $0001;    {system alert sound enabled}
```

SPECIAL CONSIDERATIONS

You can call the SndGetSysBeepState procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SndGetSysBeepState procedure are

Trap macro	Selector
_SoundDispatch	$00180008

SndSetSysBeepState

You can use the SndSetSysBeepState function to set the state of the system alert sound.

```
FUNCTION SndSetSysBeepState (sysBeepState: Integer): OSErr;
```

sysBeepState
The desired state of the system alert sound.

DESCRIPTION

You can use the `SndSetSysBeepState` function to temporarily disable the system alert sound while you play a sound and then enable the alert sound when you are done. The `sysBeepState` parameter should be set to either `sysBeepDisable` or `sysBeepEnable`.

If your application disables the system alert sound, be sure to enable it when your application gets a suspend event.

SPECIAL CONSIDERATIONS

You can call the `SndSetSysBeepState` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndSetSysBeepState` function are

Trap macro	Selector
`_SoundDispatch`	$001C0008

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	A parameter is incorrect

GetSoundHeaderOffset

You can use the `GetSoundHeaderOffset` function to get the offset from the beginning of a sound resource to the embedded sound header.

```
FUNCTION GetSoundHeaderOffset (sndHdl: Handle;
                               VAR offset: LongInt): OSErr;
```

`sndHdl` A handle to a sound resource.

`offset` On exit, the offset from the beginning of the sound resource specified by the `sndHdl` parameter to the beginning of the sound header within that sound resource.

DESCRIPTION

The `GetSoundHeaderOffset` function returns, in the `offset` parameter, the number of bytes from the beginning of the sound resource specified by the `sndHdl` parameter to the sound header that is contained within that resource. You might need this information if you want to use the address of that sound header in a sound command (such as the `soundCmd` or `bufferCmd` sound command).

The handle passed to `GetSoundHeaderOffset` does not have to be locked.

SPECIAL CONSIDERATIONS

The `GetSoundHeaderOffset` function is available only in version 3.0 and later of the Sound Manager. See "Obtaining a Pointer to a Sound Header" beginning on page 2-57 for a function you can call in earlier versions of the Sound Manager to obtain the same information.

You can call the `GetSoundHeaderOffset` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSoundHeaderOffset` function are

Trap macro	Selector
`_SoundDispatch`	$04040024

RESULT CODES

`noErr`	0	No error
`badFormat`	–206	Resource is corrupt or unusable

SEE ALSO

See Listing 2-27 on page 2-57 for an example of calling `GetSoundHeaderOffset`.

Controlling Volume Levels

You can use the `GetSysBeepVolume` and `SetSysBeepVolume` functions to get and set the volume level of the system alert sound. You can use `GetDefaultOutputVolume` and `SetDefaultOutputVolume` to get and set the default output volume for a particular output device.

IMPORTANT
These four functions are available only in Sound Manager version 3.0 and later. ▲

With all of these functions, you specify a volume with a 16-bit value, where 0 represents no volume (that is, silence) and 256 (hexadecimal $0100) represents full volume. The right and left volumes of a stereo sound are encoded as the high word and the low word, respectively, of a 32-bit value. Moreover, it's possible to overdrive a particular volume level if you need to amplify a low signal. For example, the long word $02000200 specifies a volume level of twice full volume on both the left and right channels of a stereo sound.

In addition to the four functions described in this section, Sound Manager version 3.0 introduces two new sound commands, `getVolumeCmd` and `volumeCmd`, that you can use to get and set the volume of a particular sound channel. See page 2-96 for details on these two sound commands; see "Managing Sound Volumes" beginning on page 2-31 for a code listing that uses the `volumeCmd` command.

GetSysBeepVolume

You can use the `GetSysBeepVolume` function to determine the current volume of the system alert sound.

```
FUNCTION GetSysBeepVolume (VAR level: LongInt): OSErr;
```

`level` On exit, the current volume level of the system alert sound.

DESCRIPTION

The `GetSysBeepVolume` function returns, in the `level` parameter, the current volume level of the system alert sound. The values returned in the high and low words of the `level` parameter range from 0 (silence) to $0100 (full volume).

SPECIAL CONSIDERATIONS

The `GetSysBeepVolume` function is available only in versions 3.0 and later of the Sound Manager. You can call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSysBeepVolume` function are

Trap macro	Selector
`_SoundDispatch`	$02240024

RESULT CODES

`noErr`	0	No error

SetSysBeepVolume

You can use the `SetSysBeepVolume` function to set the current volume of the system alert sound.

```
FUNCTION SetSysBeepVolume (level: LongInt): OSErr;
```

`level` The desired volume level of the system alert sound.

DESCRIPTION

The `SetSysBeepVolume` function sets the current volume level of the system alert sound. The values you can specify in the high and low words of the `level` parameter

range from 0 (silence) to $0100 (full volume). Any calls to the `SysBeep` procedure use the volume set by the most recent call to `SetSysBeepVolume`.

SPECIAL CONSIDERATIONS

The `SetSysBeepVolume` function is available only in versions 3.0 and later of the Sound Manager. You can call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetSysBeepVolume` function are

Trap macro	Selector
`_SoundDispatch`	$02280024

RESULT CODES

`noErr`	0	No error

GetDefaultOutputVolume

You can use the `GetDefaultOutputVolume` function to determine the default volume of a sound output device.

```
FUNCTION GetDefaultOutputVolume (VAR level: LongInt): OSErr;
```

`level` On exit, the default volume level of a sound output device.

DESCRIPTION

The `GetDefaultOutputVolume` function returns, in the `level` parameter, the default volume of a sound output device. The values returned in the high and low words of the `level` parameter range from 0 (silence) to $0100 (full volume).

SPECIAL CONSIDERATIONS

The `GetDefaultOutputVolume` function is available only in versions 3.0 and later of the Sound Manager. You can call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetDefaultOutputVolume` function are

Trap macro	Selector
`_SoundDispatch`	$022C0024

RESULT CODES

noErr	0	No error

SetDefaultOutputVolume

You can use the SetDefaultOutputVolume function to set the default volume of a sound output device.

```
FUNCTION SetDefaultOutputVolume (level: LongInt): OSErr;
```

level The desired default volume level of a sound output device.

DESCRIPTION

The SetDefaultOutputVolume function sets the default volume of a sound output device. The values you can specify in the high and low words of the level parameter range from 0 (silence) to $0100 (full volume).

SPECIAL CONSIDERATIONS

The SetDefaultOutputVolume function is available only in versions 3.0 and later of the Sound Manager. You can call this function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SetDefaultOutputVolume function are

Trap macro	Selector
_SoundDispatch	$02300024

RESULT CODES

noErr	0	No error

Compressing and Expanding Audio Data

You can use the procedures Comp3to1 and Comp6to1 to compress sound data. You can use the procedures Exp1to3 and Exp1to6 to expand compressed audio data.

Comp3to1

You can use the `Comp3to1` procedure to compress sound data at a ratio of 3:1.

```
PROCEDURE Comp3to1 (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                    inState: Ptr; outState: Ptr;
                    numChannels: LongInt; whichChannel: LongInt);
```

`inBuffer` A pointer to a buffer of samples to be compressed.

`outBuffer` A pointer to a buffer where the samples are to be written.

`cnt` The number of samples to compress.

`inState` A pointer to a 128-byte buffer from which the input state of the algorithm is read, or `NIL`. To initialize the algorithm, this buffer should be filled with zeros.

`outState` A pointer to a 128-byte buffer to which the output state of the algorithm is written, or `NIL`. This buffer might be the same as that specified by the `inState` parameter.

`numChannels`
The number of channels in the buffer pointed to by the `inBuffer` parameter.

`whichChannel`
The channel to compress, when `numChannels` is greater than 1. This parameter must be in the range of 1 to `numChannels`.

DESCRIPTION

The `Comp3to1` procedure compresses `cnt` samples of sound stored in the buffer specified by `inBuffer` and places the result in the buffer specified by `outBuffer`, which must be at least `cnt/3` bytes in size. The original samples can be monophonic or include multiple channels of sound, but they must be in 8-bit offset binary format. Also, if `numChannels` is greater than 1, then the noncompressed sound must be stored in interleaved format on a sample basis.

If you compress polyphonic sound, you retain only one channel of sound, which you specify in the `whichChannel` parameter. Thus, if you use the `Comp3to1` procedure to compress three-channel sound, you will have effectively compressed the sound to one-ninth its original size in bytes. To retain multiple channels of sound after compression, you must call the `Comp3to1` procedure for each channel to be compressed and then interleave the compressed sound data on a packet basis.

The `Comp3to1` procedure compresses every 48 bytes of sound data to exactly 16 bytes of compressed sound data and compresses remaining bytes to no more than one-third the original size.

You can use the `inState` and `outState` parameters to allow the MACE compression routines to preserve information about algorithms across calls. Alternatively, you may pass `NIL` state buffers and let the Sound Manager allocate the buffers internally.

SPECIAL CONSIDERATIONS

Because the `Comp3to1` procedure might allocate and dispose of memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `Comp3to1` procedure are

Trap macro	Selector
`_SoundDispatch`	$00040010

Comp6to1

You can use the `Comp6to1` procedure to compress sound data at a ratio of 6:1.

```
PROCEDURE Comp6to1 (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                    inState: Ptr; outState: Ptr;
                    numChannels: LongInt; whichChannel: LongInt);
```

`inBuffer` A pointer to a buffer of samples to be compressed.

`outBuffer` A pointer to a buffer where the samples are to be written.

`cnt` The number of samples to compress.

`inState` A pointer to a 128-byte buffer from which the input state of the algorithm is read, or `NIL`. To initialize the algorithm, this buffer should be filled with zeros.

`outState` A pointer to a 128-byte buffer to which the output state of the algorithm is written, or `NIL`. This buffer might be the same as that specified by the `inState` parameter.

`numChannels`
The number of channels in the buffer pointed to by the `inBuffer` parameter.

`whichChannel`
The channel to compress, when `numChannels` is greater than 1. This parameter must be in the range of 1 to `numChannels`.

DESCRIPTION

The `Comp6to1` procedure compresses `cnt` samples of sound stored in the buffer specified by `inBuffer` and places the result in the buffer specified by `outBuffer`, which must be at least `cnt/6` bytes in size. The `Comp6to1` procedure works much like the `Comp3to1` procedure, but compresses every 48 bytes of sound data to exactly 8 bytes of compressed sound data and compresses remaining bytes to no more than one-sixth the original size.

SPECIAL CONSIDERATIONS

Because the `Comp6to1` procedure might allocate and dispose of memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `Comp6to1` procedure are

Trap macro	Selector
`_SoundDispatch`	$000C0010

Exp1to3

You can use the `Exp1to3` procedure to expand a buffer of sound samples you previously have compressed with the `Comp3to1` procedure.

```
PROCEDURE Exp1to3 (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                   inState: Ptr; outState: Ptr;
                   numChannels: LongInt; whichChannel: LongInt);
```

`inBuffer` A pointer to a buffer of packets to be expanded.

`outBuffer` A pointer to a buffer where the expanded samples will be written.

`cnt` The number of packets to expand.

`inState` A pointer to a 128-byte buffer from which the input state of the algorithm is read, or `NIL`. To initialize the algorithm, this buffer should be filled with zeros.

`outState` A pointer to a 128-byte buffer to which the output state of the algorithm is written, or `NIL`. This buffer might be the same as that specified by the `inState` parameter.

`numChannels`
The number of channels in the buffer pointed to by the `inBuffer` parameter.

`whichChannel`
The channel to expand, when `numChannels` is greater than 1. This parameter must be in the range of 1 to `numChannels`.

DESCRIPTION

The `Exp1to3` procedure expands `cnt` packets of sound stored in the buffer specified by `inBuffer` and places the result in the buffer specified by `outBuffer`, whose size must be at least `cnt` packets * 2 bytes per packet * 3, or `cnt * 6` bytes. If `numChannels` is greater than 1, then the compressed sound must be stored in interleaved format on a packet basis.

If you expand compressed sound data that includes multiple sound channels, you retain only one channel of sound, which you specify in the `whichChannel` parameter. Thus, if you use the `Exp1to3` procedure to expand three-channel sound, the output buffer will be the same size as the input buffer since only one channel is retained. To retain multiple channels of sound after expansion, you must call the `Exp1to3` procedure for each channel to be expanded and then interleave the expanded sound data on a sample basis.

The `Exp1to3` procedure expands every packet of sampled-sound data to exactly 6 bytes.

You can use the `inState` and `outState` parameters to allow the MACE compression routines to preserve information about algorithms across calls. Alternatively, you may pass `NIL` state buffers and let the Sound Manager allocate the buffers internally.

SPECIAL CONSIDERATIONS

Because the `Exp1to3` procedure might allocate memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `Exp1to3` procedure are

Trap macro	Selector
`_SoundDispatch`	$00080010

Exp1to6

You can use the `Exp1to6` procedure to expand a buffer of sound samples you previously have compressed with the `Comp6to1` procedure.

```
PROCEDURE Exp1to6 (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                   inState: Ptr; outState: Ptr;
                   numChannels: LongInt; whichChannel: LongInt);
```

`inBuffer` A pointer to a buffer of packets to be expanded.

`outBuffer` A pointer to a buffer where the expanded samples will be written.

`cnt` The number of packets to expand.

`inState` A pointer to a 128-byte buffer from which the input state of the algorithm is read, or `NIL`. To initialize the algorithm, this buffer should be filled with zeros.

`outState` A pointer to a 128-byte buffer to which the output state of the algorithm is written, or `NIL`. This buffer might be the same as that specified by the `inState` parameter.

`numChannels`
The number of channels in the buffer pointed to by the `inBuffer` parameter.

`whichChannel`
The channel to expand, when `numChannels` is greater than 1. This parameter must be in the range of 1 to `numChannels`.

DESCRIPTION

The `Exp1to6` procedure expands `cnt` packets of sound stored in the buffer specified by `inBuffer` and places the result in the buffer specified by `outBuffer`, whose size must be at least `cnt` packets * 1 byte per packet * 6, or `cnt * 6` bytes. If `numChannels` is greater than 1, then the compressed sound must be stored in interleaved format on a packet basis. The `Exp1to6` procedure works just like the `Exp1to3` procedure, but expands 1-byte packets rather than 2-byte packets.

SPECIAL CONSIDERATIONS

Because the `Exp1to6` procedure might allocate memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `Exp1to6` procedure are

Trap macro	Selector
`_SoundDispatch`	$00100010

Managing Double Buffers

If you wish to customize the double buffering algorithm that the Sound Manager uses to manage a play from disk, you can use the `SndPlayDoubleBuffer` function. The Sound Manager's high-level play-from-disk routines make extensive use of this function.

SndPlayDoubleBuffer

The `SndPlayDoubleBuffer` function is a low-level routine that gives you maximum efficiency and control over double buffering while still maintaining compatibility with the Sound Manager.

```
FUNCTION SndPlayDoubleBuffer (chan: SndChannelPtr;
                    theParams: SndDoubleBufferHeaderPtr): OSErr;
```

`chan` A pointer to a valid sound channel.

`theParams` A pointer to a sound double buffer header record.

DESCRIPTION

The `SndPlayDoubleBuffer` function launches a low-level sound play using the information in the double buffer header record specified by `theParams`. After your application calls this function, the Sound Manager repeatedly calls the doubleback procedure you specify in the double buffer header record. The doubleback procedure then manages the filling of buffers of sound data from disk whenever one of the two buffers specified in the double buffer header record becomes exhausted.

SPECIAL CONSIDERATIONS

Because the `SndPlayDoubleBuffer` function might move memory, you should not call it at interrupt time.

You can use the `SndPlayDoubleBuffer` function only on a Macintosh computer that supports the play-from-disk routines. For information on how to determine whether a computer supports these routines, see "Testing for Multichannel Sound and Play-From-Disk Capabilities" on page 2-35.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndPlayDoubleBuffer` function are

Trap macro	Selector
`_SoundDispatch`	$00200008

RESULT CODES

`noErr`	0	No error
`badChannel`	–205	Channel is corrupt or unusable

SEE ALSO

For information on the format of a doubleback procedure, see "Doubleback Procedures" on page 2-153.

Performing Unsigned Fixed-Point Arithmetic

This section describes the `UnsignedFixMulDiv` function provided by the Sound Manager that you can use to perform multiplication and division on unsigned fixed-point numbers.

UnsignedFixMulDiv

You can use the `UnsignedFixMulDiv` function to perform multiplications and divisions on unsigned fixed-point numbers. You'll typically use it to calculate sample rates.

```
FUNCTION UnsignedFixMulDiv (value: UnsignedFixed;
                            multiplier: UnsignedFixed;
                            divisor: UnsignedFixed):
                            UnsignedFixed;
```

`value` The value to be multiplied and divided.

`multiplier`
The multiplier to be applied to the value in the `value` parameter.

`divisor` The divisor to be applied to the value in the `value` parameter.

DESCRIPTION

The `UnsignedFixMulDiv` function returns the fixed-point number that is the value of the `value` parameter, multiplied by the value in the `multiplier` parameter and divided by the value in the `divisor` parameter. Note that `UnsignedFixMulDiv` performs both operations before returning. If you want to perform only a multiplication or only a division, pass the value $00010000 for whichever parameter you want to ignore. For example, to determine the sample rate that is twice that of the 22 kHz rate, you can use `UnsignedFixMulDiv` as follows:

```
myNewRate := UnsignedFixMulDiv(rate22kHz, $00020000, $00010000);
```

Similarly, to determine the sample rate that is half that of the 44 kHz rate, you can use `UnsignedFixMulDiv` as follows:

```
myNewRate := UnsignedFixMulDiv(rate44kHz, $00010000, $00020000);
```

SPECIAL CONSIDERATIONS

The `UnsignedFixMulDiv` function is available only in versions 3.0 and later of the Sound Manager.

Linking Modifiers to Sound Channels

Early versions of the Sound Manager allowed application developers to use modifiers to alter sound commands before being processed by the Sound Manager. The Sound Manager no longer supports this capability. `SndAddModifier` is documented here for completeness only.

SndAddModifier

The Sound Manager previously used the `SndAddModifier` function to link modifiers to sound channels.

```
FUNCTION SndAddModifier (chan: SndChannelPtr; modifier: ProcPtr;
                         id: Integer; init: LongInt): OSErr;
```

`chan`	A pointer to a valid sound channel.
`modifier`	A pointer to a modifier function to be added to the sound channel specified by `chan`. This field is obsolete.
`id`	The resource ID of the modifier to be linked to the sound channel.
`init`	The initialization parameters for the sound channel specified by `chan`.

DESCRIPTION

The `SndAddModifier` function installs a modifier into an open channel specified in the `chan` parameter. The `modifier` parameter should be `NIL`, and the `id` parameter is the resource ID of the modifier to be linked to the sound channel. `SndAddModifier` causes the Sound Manager to load the specified `'snth'` resource, lock it in memory, and link it to the channel specified.

IMPORTANT

The `SndAddModifier` function is for internal Sound Manager use only. You should not call it in your application. ▲

The only supported use of the `SndAddModifier` function is to change the data type associated with a sound channel. For example, you can pass the constant `sampledSynth` in the `id` parameter to reconfigure a sound channel for sampled-sound data. You should, however, set a sound channel's data type when you call `SndNewChannel`, not by calling `SndAddModifier`.

SPECIAL CONSIDERATIONS

You should not use the `SndAddModifier` function.

RESULT CODES

`noErr`	0	No error
`resProblem`	–204	Problem loading the resource
`badChannel`	–205	Channel is corrupt or unusable

SEE ALSO

To modify sampled-sound data immediately before the Sound Manager plays it, you can customize double buffering routines so that your application can modify sampled-sound

data when it fills a buffer of sound data for the Sound Manager to play. For more information, see "Using Double Buffers" on page 2-68.

To change the initialization options for a sound channel, you can use the `reInitCmd` command. For a description of that command, see "Sound Command Numbers" beginning on page 2-92.

Application-Defined Routines

The Sound Manager allows you to define a completion routine that execute when a play from disk finishes executing, a callback procedure that executes whenever your application issues the `callBackCmd` command, and a doubleback procedure that you must define if you wish to customize the double buffering of data during a play from disk.

Completion Routines

You can specify a completion routine as the seventh parameter to the `SndStartFilePlay` function. The completion routine executes when the sound file finishes playing (unless sound play was stopped by the `SndStopFilePlay` function).

MyCompletionRoutine

A Sound Manager completion routine has the following syntax:

```
PROCEDURE MyFilePlayCompletionRoutine (chan: SndChannelPtr);
```

`chan` A pointer to the sound channel on which a play from disk has completed.

DESCRIPTION

The Sound Manager executes your completion routine when a play from disk on the channel specified by the `chan` parameter finishes. You might use the completion routine to set a global flag that alerts the application that it must dispose of the sound channel.

SPECIAL CONSIDERATIONS

A completion routine is called at interrupt time. It must not make any calls to the Memory Manager, either directly or indirectly. If your completion routine needs to access your application's global variables, you must ensure that register A5 contains your application's A5. (You can use the `userInfo` field of the sound channel pointed to by the `chan` parameter to pass that value to your completion routine.)

ASSEMBLY-LANGUAGE INFORMATION

Because this routine is called at interrupt time, it must preserve all registers other than A0–A1 and D0–D2.

SEE ALSO

For information on how you can use completion routines to help manage an asynchronous play from disk, see "Managing an Asynchronous Play From Disk" on page 2-52.

Callback Procedures

You can specify a callback procedure as the fourth parameter to the `SndNewChannel` function. The callback procedure executes whenever the Sound Manager processes a `callBackCmd` command for the channel.

MyCallbackProcedure

A callback procedure has the following syntax:

```
PROCEDURE MyCallbackProcedure (theChan: SndChannelPtr;
                               theCmd: SndCommand);
```

`theChan` A pointer to the sound channel on which a `callBackCmd` command was issued.

`theCmd` The sound command record in which a `callBackCmd` command was issued.

DESCRIPTION

The Sound Manager executes the callback procedure associated with a sound channel whenever it processes a `callBackCmd` command for the channel. You can use a callback procedure to set a global flag that alerts the application that it must dispose of the sound channel. Or, you can use a callback procedure so that your application can synchronize a series of sound commands with other actions.

SPECIAL CONSIDERATIONS

A callback procedure is called at interrupt time. It must not make any calls to the Memory Manager, either directly or indirectly. If your callback procedure needs to access your application's global variables, you must ensure that register A5 contains your application's A5. (You can use the `userInfo` field of the sound channel pointed to by the `theChan` parameter or the `param2` field of the sound command specified in the `theCmd` parameter to pass that value to your callback procedure.)

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A1 and D0–D2.

SEE ALSO

For information on how you can use callback procedures when playing sound asynchronously, see "Using Callback Procedures" on page 2-47.

Doubleback Procedures

If you wish to customize the double buffering of sound during a play from disk, you must use the `SndPlayDoubleBuffer` function and define a doubleback procedure. Doubleback procedures also give you the power to modify sampled-sound data immediately before the Sound Manager plays it.

MyDoubleBackProc

A doubleback procedure has the following syntax:

```
PROCEDURE MyDoubleBackProc (chan: SndChannelPtr;
                            exhaustedBuffer: SndDoubleBufferPtr);
```

`chan` A pointer to a sound channel on which a play from disk is executing.

`exhaustedBuffer`
A pointer to a sound double buffer record

DESCRIPTION

The Sound Manager calls the doubleback procedure associated with a play from disk whenever the Sound Manager has exhausted the buffer. As the doubleback procedure refills the buffer, the Sound Manager plays the other buffer. Your application might also call the doubleback procedure twice to fill both buffers before the initial call to `SndPlayDoubleBuffer` function.

When your doubleback procedure is called, it must

- fill the buffer specified in the `exhaustedBuffer` parameter with the next set of sound frames that the Sound Manager must play
- set the `dbNumFrames` field of the sound double buffer record to the number of frames in the buffer
- set the `dbBufferReady` bit of the `dbFlags` field of the sound double buffer record

If your doubleback procedure fills the buffer with the last frames of sound that need to be played, then your procedure should set the `dbLastBuffer` bit of the `dbFlags` field of the sound double buffer record.

Your doubleback procedure might fill the buffer with data from any of several sources. For example, the doubleback procedure might compute the data, copy it from elsewhere in RAM, or read it from disk. A doubleback procedure can also read data from disk and then modify the data. This might be useful, for example, if you would like the Sound Manager to be able to play sampled-sound data stored in 16-bit binary offset format. Your doubleback procedure could translate the data to the 8-bit binary offset format that the Sound Manager can read before placing it in the buffer.

SPECIAL CONSIDERATIONS

A doubleback procedure is called at interrupt time. It must not make any calls to the Memory Manager, either directly or indirectly. If your callback procedure needs to access your application's global variables, you must ensure that register A5 contains your application's A5. (You can use one of the two long integers in the `dbUserInfo` field of the sound double buffer record specified by the `exhaustedBuffer` parameter to pass that value to your callback procedure.)

ASSEMBLY-LANGUAGE INFORMATION

Because a doubleback procedure is called at interrupt time, it must preserve all registers other than A0–A1 and D0–D2.

SEE ALSO

For an example of how you might use doubleback procedures, see "Using Double Buffers" on page 2-68.

Resources

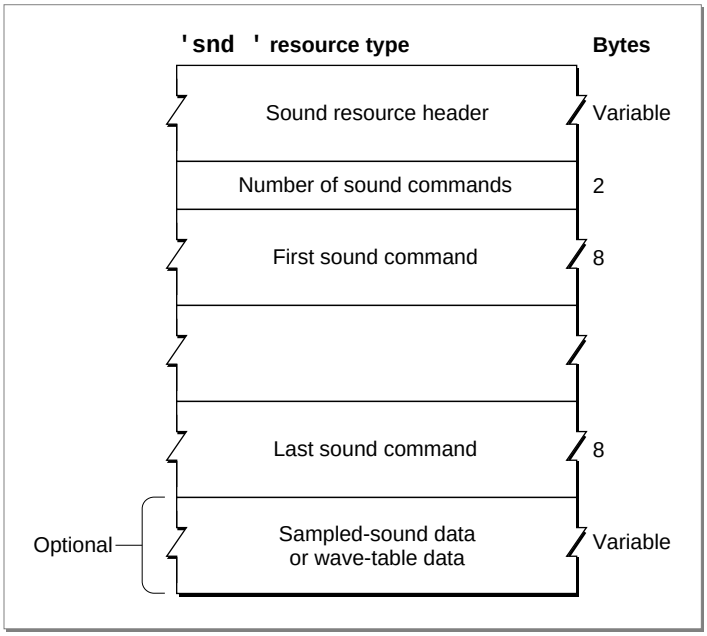
This section describes the structure of format 1 and format 2 sound resources. For a more complete discussion of the structure of sound resources, see "Sound Resources" on page 2-74.

The Sound Resource

You can store sound commands and sound data as a resource with the resource type `'snd '`. Resource IDs from 0 to 8191 are reserved by Apple Computer, Inc. You may use all other resource IDs for your `'snd '` resources.

You can use the `GetResource` function to search all open resource files for the first `'snd '` resource type with the given ID. The `'snd '` resource type defines a sound resource. Figure 2-8 shows the structure of a sound resource.

Figure 2-8 The 'snd ' resource type



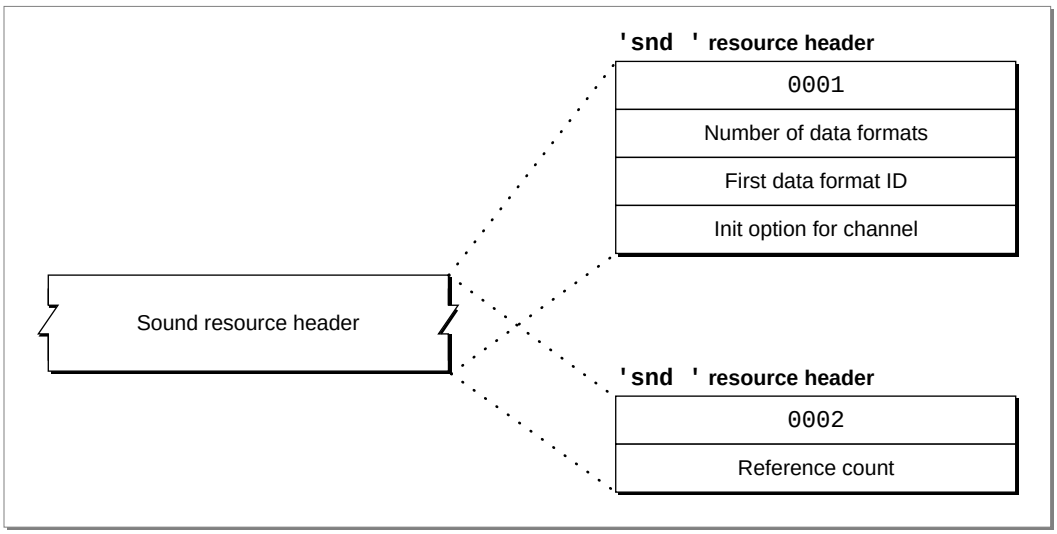
Often, you can create a sound resource simply by using the `SndRecord` function, documented in the chapter "Introduction to Sound on the Macintosh" in this book. However, you can also define a sound resource manually. This is especially useful for sound resources that are simply series of sound commands and contain no sampled-sound data. Also, you might construct a sound resource that contains wave-table data manually. A sound resource contains the following elements:

- Sound resource header. The gives information about the format of a sound resource, as explained below.
- Number of sound commands. Following the sound resource header is a word indicating the number of sound commands contained in the resource.
- Sound commands. Each sound command is 8 bytes, which includes 2 bytes that identify the command, 2 bytes for the command's first parameter, and 4 bytes for the command's second parameter. When a sound command contained in an `'snd '` resource has associated sound data, the high bit (defined by the `dataOffsetFlag` constant) should be set. This tells the Sound Manager that the value in the second parameter is an offset from the beginning of the resource and not a pointer to a memory location.
- Sound data. For a format 1 `'snd '` resource, this field might contain wave-table data or a sampled sound header that includes sampled-sound data. For a format 2 `'snd '` resource, this field should contain a sampled sound header that includes sampled-sound data.

The format of the sound resource header differs depending on whether the `'snd '` resource is format 1 or format 2. Figure 2-9 illustrates the formats of the two types of

sound resource header. Both sound headers begin with a format field, which defines the format of the sound resource as either $0001 or $0002.

Figure 2-9 The sound resource header



- Format 1 sound resource header. For format 1 `'snd '` resources, the sound resource header includes a word that indicates the number of data types to be sent to the sound channel. Because a sound channel cannot play more than one type of sound data, you should typically specify either $00 or $01 in this field. If you specify $01 or more, then the sound resource header contains both a word specifying the data type and a long word specifying the initialization options for each data type.
- Format 2 sound resource header. For format 2 `'snd '` resources, the sound resource header next includes a single word that the Sound Manager ignores. This word is known as the reference count field. Your application can use this field as it pleases.

Summary of the Sound Manager

Pascal Summary

Constants

```
CONST
   {Gestalt sound attributes selector and response bits}
   gestaltSoundAttr           = 'snd ';{sound attributes selector}
   gestaltStereoCapability    = 0;     {built-in hw can play stereo sounds}
   gestaltStereoMixing        = 1;     {built-in hw mixes stereo to mono}
   gestaltSoundIOMgrPresent   = 3;     {sound input routines available}
   gestaltBuiltInSoundInput   = 4;     {built-in input hw available}
   gestaltHasSoundInputDevice = 5;     {sound input device available}
   gestaltPlayAndRecord       = 6;     {built-in hw can play while recording}
   gestalt16BitSoundIO        = 7;     {built-in hw can handle 16-bit data}
   gestaltStereoInput         = 8;     {built-in hw can record stereo sounds}
   gestaltLineLevelInput      = 9;     {built-in input hw needs line level}
   gestaltSndPlayDoubleBuffer = 10;    {play from disk routines available}
   gestaltMultiChannels       = 11;    {multiple channels of sound supported}
   gestalt16BitAudioSupport   = 12;    {16-bit audio data supported}

   {channel initialization parameters}
   initChanLeft               = $0002; {left stereo channel}
   initChanRight              = $0003; {right stereo channel}
   waveInitChannel0           = $0004; {wave-table channel 0}
   waveInitChannel1           = $0005; {wave-table channel 1}
   waveInitChanne12           = $0006; {wave-table channel 2}
   waveInitChannel3           = $0007; {wave-table channel 3}
   initMono                   = $0080; {monophonic channel}
   initStereo                 = $00C0; {stereo channel}
   initMACE3                  = $0300; {3:1 compression}
   initMACE6                  = $0400; {6:1 compression}
   initNoInterp               = $0004; {no linear interpolation}
   initNoDrop                 = $0008; {no drop-sample conversion}
```

```
{masks for channel attributes}
initPanMask             = $0003;    {mask for right/left pan values}
initSRateMask           = $0030;    {mask for sample rate values}
initStereoMask          = $00C0;    {mask for mono/stereo values}
initCompMask            = $FF00;    {mask for compression IDs}

{sound data types}
squareWaveSynth         = 1;        {square-wave data}
waveTableSynth          = 3;        {wave-table data}
sampledSynth            = 5;        {sampled-sound data}

{sound command numbers}
nullCmd                 = 0;        {do nothing}
quietCmd                = 3;        {stop a sound that is playing}
flushCmd                = 4;        {flush a sound channel}
reInitCmd               = 5;        {reinitialize a sound channel}
waitCmd                 = 10;       {suspend processing in a channel}
pauseCmd                = 11;       {pause processing in a channel}
resumeCmd               = 12;       {resume processing in a channel}
callBackCmd             = 13;       {execute a callback procedure}
syncCmd                 = 14;       {synchronize channels}
availableCmd            = 24;       {see if initialization options }
                                    { are supported}
versionCmd              = 25;       {determine version}
totalLoadCmd            = 26;       {report total CPU load}
loadCmd                 = 27;       {report CPU load for a new channel}
freqDurationCmd         = 40;       {play a note for a duration}
restCmd                 = 41;       {rest a channel for a duration}
freqCmd                 = 42;       {change the pitch of a sound
ampCmd                  = 43;       {change the amplitude of a sound}
timbreCmd               = 44;       {change the timbre of a sound}
getAmpCmd               = 45;       {get the amplitude of a sound}
volumeCmd               = 46;       {set volume}
getVolumeCmd            = 47;       {get volume}
waveTableCmd            = 60;       {install a wave table as a voice}
soundCmd                = 80;       {install a sampled sound as a voice}
bufferCmd               = 81;       {play a sampled sound}
rateCmd                 = 82;       {set the pitch of a sampled sound}
getRateCmd              = 85;       {get the pitch of a sampled sound}

{sampled sound header encoding options}
stdSH                   = $00;      {standard sound header}
extSH                   = $FF;      {extended sound header}
cmpSH                   = $FE;      {compressed sound header}
```

```
{size of data structures}
stdQLength              = 128;      {default size of standard sound }
                                    { channel}

{sound resource formats}
firstSoundFormat        = $0001;    {format 1 'snd ' resource}
secondSoundFormat       = $0002;    {format 2 'snd ' resource}

{sound command mask}
dataOffsetFlag          = $8000;    {sound command data offset bit}

{system beep states}
sysBeepDisable          = $0000;    {system alert sound disabled}
sysBeepEnable           = $0001;    {system alert sound enabled}

{values for the unitType field in AudioSelection}
unitTypeSeconds         = $0000;    {seconds}
unitTypeNoSelection     = $FFFF;    {no selection}

{double buffer status flags}
dbBufferReady           = $00000001;{double buffer is filled}
dbLastBuffer            = $00000004;{last double buffer to play}

{values for the compressionID field of CmpSoundHeader}
variableCompression     = -2;       {variable-ratio compression}
fixedCompression        = -1;       {fixed-ratio compression}
notCompressed           = 0;        {noncompressed samples}
threeToOne              = 3;        {3:1 compressed samples}
sixToOne                = 4;        {6:1 compressed samples}

{values for the packetSize field of CmpSoundHeader}
sixToOnePacketSize      = 8;        {packet size in bits for 6:1}
threeToOnePacketSize    = 16;       {packet size in bits for 3:1}

{compression names and types}
NoneName                = 'not compressed';
ACE2to1Name             = 'ACE 2-to-1';
ACE8to3Name             = 'ACE 8-to-3';
MACE3to1Name            = 'MACE 3-to-1';
MACE6to1Name            = 'MACE 6-to-1';
NoneType                = 'NONE';
ACE2Type                = 'ACE2';
ACE8Type                = 'ACE8';
MACE3Type               = 'MAC3';
MACE6Type               = 'MAC6'
```

```
{IDs for AIFF and AIFF-C files}
AIFFID                  = 'AIFF';   {AIFF file}
AIFCID                  = 'AIFC';   {AIFF-C file}

{IDs for AIFF and AIFF-C file chunks}
FormID                  = 'FORM';   {ID for Form Chunk}
FormatVersionID         = 'FVER';   {ID for Format Version Chunk}
CommonID                = 'COMM';   {ID for Common Chunk}
SoundDataID             = 'SSND';   {ID for Sound Data Chunk}
MarkerID                = 'MARK';   {ID for Marker Chunk}
InstrumentID            = 'INST';   {ID for Instrument Chunk}
MIDIDataID              = 'MIDI';   {ID for MIDI Data Chunk}
AudioRecordingID        = 'AESD';   {ID for Recording Chunk}
ApplicationSpecificID   = 'APPL';   {ID for Application Chunk}
CommentID               = 'COMT';   {ID for Comment Chunk}
NameID                  = 'NAME';   {ID for Name Chunk}
AuthorID                = 'AUTH';   {ID for Author Chunk}
CopyrightID             = '(c) ';   {ID for Copyright Chunk}
AnnotationID            = 'ANNO';   {ID for Annotation Chunk}

{version of AIFC format specification}
AIFCVersion1            = $A2805140;{date of version creation}

{MIDI note value for middle C}
kMiddleC                = 60;

{ratio between frequencies of MIDI note values}
twelfthRootTwo          = 1.05946309434;

{standard sampling rates}
rate44khz               = $AC440000;   {44100.00000 in fixed-point}
rate22khz               = $56EE8BA3;   {22254.54545 in fixed-point}
rate22050hz             = $56220000;   {22050.00000 in fixed-point}
rate11khz               = $2B7745D1;   {11127.27273 in fixed-point}
rate11025hz             = $2B110000;   {11025.00000 in fixed-point}

{constant for synth parameter of SndNewChannel}
kUseOptionalOutputDevice      = -1;

{volumes}
kFullVolume                   = $0100;
kNoVolume                     = 0;
```

```
{development stages}
developStage            = $20;          {prealpha release}
alphaStage              = $40;          {alpha release}
betaStage               = $60;          {beta release}
finalStage              = $80;          {final release}

{sizes of data buffers}
stateBlockSize          = 64;           {size of state block buffer}
leftOverBlockSize       = 32;           {size of leftover block buffer}
```


Data Types

Unsigned Fixed-Point Numbers

```
TYPE
   UnsignedFixed = LongInt;             {unsigned fixed-point number}
```

Times

```
TYPE
   Time = LongInt;                      {in half-milliseconds}
```

Sound Command Record

```
   SndCommand =
   PACKED RECORD
      cmd:              Integer;        {command number}
      param1:           Integer;        {first parameter}
      param2:           LongInt;        {second parameter}
   END;
```

Audio Selection Record

```
   AudioSelection =
   PACKED RECORD
      unitType:         LongInt;        {type of time unit}
      selStart:         Fixed;          {starting point of selection}
      selEnd:           Fixed;          {ending point of selection}
   END;
   AudioSelectionPtr = ^AudioSelection;
```

Sound Channel Status Record

```
SCStatus =
RECORD
   scStartTime:         Fixed;      {starting time for play from disk}
   scEndTime:           Fixed;      {ending time for play from disk}
   scCurrentTime:       Fixed;      {current time for play from disk}
   scChannelBusy:       Boolean;    {TRUE if channel is processing cmds}
   scChannelDisposed:   Boolean;    {reserved}
   scChannelPaused:     Boolean;    {TRUE if play from disk is paused}
   scUnused:            Boolean;    {unused}
   scChannelAttributes: LongInt;    {attributes of this channel}
   scCPULoad:           LongInt;    {CPU load for this channel}
END;
SCStatusPtr = ^SCStatus;
```

Sound Manager Status Record

```
SMStatus =
PACKED RECORD
   smMaxCPULoad:     Integer;       {maximum load on all channels}
   smNumChannels:    Integer;       {number of allocated channels}
   smCurCPULoad:     Integer;       {current load on all channels}
END;
SMStatusPtr = ^SMStatus;
```

Sound Channel Record

```
SndChannel =
PACKED RECORD
   nextChan:         SndChannelPtr; {pointer to next channel}
   firstMod:         Ptr;           {used internally}
   callBack:         ProcPtr;       {pointer to callback procedure}
   userInfo:         LongInt;       {free for application's use}
   wait:             LongInt;       {used internally}
   cmdInProgress:    SndCommand;    {used internally}
   flags:            Integer;       {used internally}
   qLength:          Integer;       {used internally}
   qHead:            Integer;       {used internally}
   qTail:            Integer;       {used internally}
   queue:            ARRAY[0..stdQLength-1] OF SndCommand;
END;
SndChannelPtr = ^SndChannel;
```

Sound Header Record

```
SoundHeader =
PACKED RECORD
   samplePtr:        Ptr;           {if NIL, samples in sampleArea}
   length:           LongInt;       {number of samples in array}
   sampleRate:       Fixed;         {sample rate}
   loopStart:        LongInt;       {loop point beginning}
   loopEnd:          LongInt;       {loop point ending}
   encode:           Byte;          {sample's encoding option}
   baseFrequency:    Byte;          {base frequency of sample}
   sampleArea:       PACKED ARRAY[0..0] OF Byte;
END;
SoundHeaderPtr = ^SoundHeader;
```

Extended Sound Header Record

```
ExtSoundHeader =
PACKED RECORD
   samplePtr:        Ptr;           {if NIL, samples in sampleArea}
   numChannels:      LongInt;       {number of channels in sample}
   sampleRate:       Fixed;         {rate of original sample}
   loopStart:        LongInt;       {loop point beginning}
   loopEnd:          LongInt;       {loop point ending}
   encode:           Byte;          {sample's encoding option}
   baseFrequency:    Byte;          {base frequency of sample}
   numFrames:        LongInt;       {total number of frames}
   AIFFSampleRate:   Extended80;    {rate of original sample}
   markerChunk:      Ptr;           {reserved}
   instrumentChunks: Ptr;           {pointer to instrument info}
   AESRecording:     Ptr;           {pointer to audio info}
   sampleSize:       Integer;       {number of bits per sample}
   futureUse1:       Integer;       {reserved}
   futureUse2:       LongInt;       {reserved}
   futureUse3:       LongInt;       {reserved}
   futureUse4:       LongInt;       {reserved}
   sampleArea:       PACKED ARRAY[0..0] OF Byte;
END;
ExtSoundHeaderPtr = ^ExtSoundHeader;
```

Compressed Sound Header Record

```
CmpSoundHeader =
PACKED RECORD
   samplePtr:        Ptr;           {if NIL, samples in sampleArea}
   numChannels:      LongInt;       {number of channels in sample}
   sampleRate:       Fixed;         {rate of original sample}
   loopStart:        LongInt;       {loop point beginning}
   loopEnd:          LongInt;       {loop point ending}
   encode:           Byte;          {sample's encoding option}
   baseFrequency:    Byte;          {base freq. of original sample}
   numFrames:        LongInt;       {length of sample in frames}
   AIFFSampleRate:   Extended80;    {rate of original sample}
   markerChunk:      Ptr;           {reserved}
   format:           OSType;        {data format type}
   futureUse2:       LongInt;       {reserved}
   stateVars:        StateBlockPtr; {pointer to StateBlock}
   leftOverSamples:  LeftOverBlockPtr;
                                    {pointer to LeftOverBlock}
   compressionID:    Integer;       {ID of compression algorithm}
   packetSize:       Integer;       {number of bits per packet}
   snthID:           Integer;       {unused}
   sampleSize:       Integer;       {bits in each sample point}
   sampleArea:       PACKED ARRAY[0..0] OF Byte;
END;
CmpSoundHeaderPtr = ^CmpSoundHeader;
```

Sound Double Buffer Header Record

```
SndDoubleBufferHeader =
PACKED RECORD
   dbhNumChannels:   Integer;       {number of sound channels}
   dbhSampleSize:    Integer;       {sample size, if noncompressed}
   dbhCompressionID: Integer;       {ID of compression algorithm}
   dbhPacketSize:    Integer;       {number of bits per packet}
   dbhSampleRate:    Fixed;         {sample rate}
   dbhBufferPtr:     ARRAY[0..1] OF SndDoubleBufferPtr;
                                    {pointers to SndDoubleBuffer}
   dbhDoubleBack:    ProcPtr;       {pointer to doubleback procedure}
END;
SndDoubleBufferHeaderPtr = ^SndDoubleBufferHeader;
```

```
SndDoubleBufferHeader2 =
PACKED RECORD
   dbhNumChannels:   Integer;       {number of sound channels}
   dbhSampleSize:    Integer;       {sample size, if noncompressed}
   dbhCompressionID: Integer;       {ID of compression algorithm}
   dbhPacketSize:    Integer;       {number of bits per packet}
   dbhSampleRate:    Fixed;         {sample rate}
   dbhBufferPtr:     ARRAY[0..1] OF SndDoubleBufferPtr;
                                    {pointers to SndDoubleBuffer}
   dbhDoubleBack:    ProcPtr;       {pointer to doubleback procedure}
   dbhFormat:        OSType;        {signature of codec}
END;
SndDoubleBufferHeaderPtr2 = ^SndDoubleBufferHeader2;
```

Sound Double Buffer Record

```
SndDoubleBuffer =
PACKED RECORD
   dbNumFrames:      LongInt;       {number of frames in buffer}
   dbFlags:          LongInt;       {buffer status flags}
   dbUserInfo:       ARRAY[0..1] OF LongInt;
                                    {for application's use}
   dbSoundData:      PACKED ARRAY[0..0] OF Byte;
                                    {array of data}
END;
SndDoubleBufferPtr = ^SndDoubleBuffer;
```

Chunk Header

```
ID = LongInt;                       {chunk ID type}

ChunkHeader =
RECORD
   ckID:             ID;            {chunk type ID}
   ckSize:           LongInt;       {number of bytes of data}
END;
```

Form Chunk

```
ContainerChunk =
RECORD
   ckID:            ID;           {'FORM'}
   ckSize:          LongInt;      {number of bytes of data}
   formType:        ID;           {type of file}
END;
```

Format Version Chunk

```
FormatVersionChunk =
RECORD
   ckID:            ID;           {'FVER'}
   ckSize:          LongInt;      {4 bytes}
   timestamp:       LongInt;      {date of format version}
END;
```

Common Chunk

```
CommonChunk =
RECORD
   ckID:            ID;           {'COMM'}
   ckSize:          LongInt;      {18 bytes}
   numChannels:     Integer;      {number of channels}
   numSampleFrames: LongInt;      {number of sample frames}
   sampleSize:      Integer;      {number of bits per sample}
   sampleRate:      Extended;     {number of frames per second}
END;
```

Extended Common Chunk

```
ExtCommonChunk =
RECORD
   ckID:            ID;           {'COMM'}
   ckSize:          LongInt;      {22 bytes + compression name}
   numChannels:     Integer;      {number of channels}
   numSampleFrames: LongInt;      {number of sample frames}
   sampleSize:      Integer;      {number of bits per sample}
   sampleRate:      Extended;     {number of frames per second}
   compressionType: ID;           {compression type ID}
   compressionName: PACKED ARRAY[0..0] OF Byte;
                                  {compression type name}
END;
```

Sound Data Chunk

```
SoundDataChunk =
RECORD
   ckID:       ID;         {'SSND'}
   ckSize:     LongInt;    {size of chunk data}
   offset:     LongInt;    {offset to sound data}
   blockSize:  LongInt;    {size of alignment blocks}
END;
```

Version Record

```
NumVersion =
PACKED RECORD
CASE INTEGER OF
 0:
   (majorRev:        SignedByte;    {major revision level in BCD}
   minorAndBugRev:   SignedByte;    {minor revision level}
   stage:            SignedByte;    {development stage}
   nonRelRev:        SignedByte);   {nonreleased revision level}
 1:
   (version:         LongInt);      {all 4 fields together}
END;
```

Leftover Block

```
LeftOverBlock =
RECORD
   count:         LongInt;
   sampleArea:    PACKED ARRAY[0..leftOverBlockSize - 1] OF Byte;
END;
LeftOverBlockPtr = ^LeftOverBlock;
```

State Block

```
StateBlock =
RECORD
   stateVar:      ARRAY[0..stateBlockSize - 1] OF Integer;
END;
StateBlockPtr = ^StateBlock;
```

Sound Manager Routines

Playing Sound Resources

```
PROCEDURE SysBeep           (duration: Integer);
FUNCTION SndPlay            (chan: SndChannelPtr; sndHdl: Handle;
                             async: Boolean): OSErr;
```

Playing From Disk

```
FUNCTION SndStartFilePlay   (chan: SndChannelPtr; fRefNum: Integer;
                             resNum: Integer; bufferSize: LongInt;
                             theBuffer: Ptr;
                             theSelection: AudioSelectionPtr;
                             theCompletion: ProcPtr; async: Boolean): OSErr;
FUNCTION SndPauseFilePlay   (chan: SndChannelPtr): OSErr;
FUNCTION SndStopFilePlay    (chan: SndChannelPtr; quietNow: Boolean): OSErr;
```

Allocating and Releasing Sound Channels

```
FUNCTION SndNewChannel      (VAR chan: SndChannelPtr; synth: Integer;
                             init: LongInt; userRoutine: ProcPtr): OSErr;
FUNCTION SndDisposeChannel  (chan: SndChannelPtr; quietNow: Boolean): OSErr;
```

Sending Commands to a Sound Channel

```
FUNCTION SndDoCommand       (chan: SndChannelPtr; cmd: SndCommand;
                             noWait: Boolean): OSErr;
FUNCTION SndDoImmediate     (chan: SndChannelPtr; cmd: SndCommand): OSErr;
```

Obtaining Information

```
FUNCTION SndSoundManagerVersion
                            : NumVersion;
FUNCTION MACEVersion        : NumVersion;
FUNCTION SndControl         (id: Integer; VAR cmd: SndCommand): OSErr;
FUNCTION SndChannelStatus   (chan: SndChannelPtr; theLength: Integer;
                             theStatus: SCStatusPtr): OSErr;
FUNCTION SndManagerStatus   (theLength: Integer; theStatus: SMStatusPtr):
                             OSErr;
PROCEDURE SndGetSysBeepState
                            (VAR sysBeepState: Integer);
FUNCTION SndSetSysBeepState
                            (sysBeepState: Integer): OSErr;
```

```
FUNCTION GetSoundHeaderOffset
                            (sndHdl: Handle; VAR offset: LongInt): OSErr;
```

Controlling Volume Levels

```
FUNCTION GetSysBeepVolume   (VAR level: LongInt): OSErr;
FUNCTION SetSysBeepVolume   (level: LongInt): OSErr;
FUNCTION GetDefaultOutputVolume
                            (VAR level: LongInt): OSErr;
FUNCTION SetDefaultOutputVolume
                            (level: LongInt): OSErr;
```

Compressing and Expanding Audio Data

```
PROCEDURE Comp3to1          (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                             inState: Ptr; outState: Ptr;
                             numChannels: LongInt; whichChannel: LongInt);
PROCEDURE Comp6to1          (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                             inState: Ptr; outState: Ptr;
                             numChannels: LongInt; whichChannel: LongInt);
PROCEDURE Exp1to3           (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                             inState: Ptr; outState: Ptr;
                             numChannels: LongInt; whichChannel: LongInt);
PROCEDURE Exp1to6           (inBuffer: Ptr; outBuffer: Ptr; cnt: LongInt;
                             inState: Ptr; outState: Ptr;
                             numChannels: LongInt; whichChannel: LongInt);
```

Managing Double Buffers

```
FUNCTION SndPlayDoubleBuffer
                            (chan: SndChannelPtr;
                             theParams: SndDoubleBufferHeaderPtr): OSErr;
```

Performing Unsigned Fixed-Point Arithmetic

```
FUNCTION UnsignedFixMulDiv  (value: UnsignedFixed;
                             multiplier: UnsignedFixed;
                             divisor: UnsignedFixed): UnsignedFixed;
```

Linking Modifiers to Sound Channels

```
FUNCTION SndAddModifier     (chan: SndChannelPtr; modifier: ProcPtr;
                             id: Integer; init: LongInt): OSErr;
```

Application-Defined Routines

```
PROCEDURE MyFilePlayCompletionRoutine
                            (chan: SndChannelPtr);
PROCEDURE MyCallback        (chan: SndChannelPtr; cmd: SndCommand);
PROCEDURE MyDoubleBackProc  (chan: SndChannelPtr;
                             doubleBufferPtr: SndDoubleBufferPtr);
```

C Summary

Constants

```
/*Gestalt sound attributes selector and response bits*/
#define gestaltSoundAttr    'snd '  /*sound attributes selector*/

enum {
   gestaltStereoCapability    = 0,  /*built-in hw can play stereo sounds*/
   gestaltStereoMixing        = 1,  /*built-in hw mixes stereo to mono*/
   gestaltSoundIOMgrPresent   = 3,  /*sound input routines available*/
   gestaltBuiltInSoundInput   = 4,  /*built-in input hw available*/
   gestaltHasSoundInputDevice = 5,  /*sound input device available*/
   gestaltPlayAndRecord       = 6,  /*built-in hw can play while recording*/
   gestalt16BitSoundIO        = 7,  /*built-in hw can handle 16-bit data*/
   gestaltStereoInput         = 8,  /*built-in hw can record stereo sounds*/
   gestaltLineLevelInput      = 9,  /*built-in input hw needs line level*/
   gestaltSndPlayDoubleBuffer = 10, /*play from disk routines available*/
   gestaltMultiChannels       = 11, /*multiple channels of sound supported*/
   gestalt16BitAudioSupport   = 12  /*16-bit audio data supported*/
};

/*channel initialization parameters*/
enum {
   initChanLeft            = 0x0002,   /*left stereo channel*/
   initChanRight           = 0x0003,   /*right stereo channel*/
   initMono                = 0x0080,   /*monophonic channel*/
   initStereo              = 0x00C0,   /*stereo channel*/
   initMACE3               = 0x0300,   /*3:1 compression*/
   initMACE6               = 0x0400,   /*6:1 compression*/
   initNoInterp            = 0x0004,   /*no linear interpolation*/
   initNoDrop              = 0x0008    /*no drop-sample conversion*/
};
```

```
/*wave channel initialization parameters*/
enum {
   waveInitChannel0        = 0x04,     /*wave-table channel 0*/
   waveInitChannel1        = 0x05,     /*wave-table channel 1*/
   waveInitChannel2        = 0x06,     /*wave-table channel 2*/
   waveInitChannel3        = 0x07,     /*wave-table channel 3*/
   waveInitChannelMask     = 0x07      /*mask for wave-table parameters*/
};

/*masks for channel attributes*/
enum {
   initPanMask             = 0x0003,   /*mask for left/right pan values*/
   initSRateMask           = 0x0030,   /*mask for sample rate values*/
   initStereoMask          = 0x00C0,   /*mask for mono/stereo values*/
   initCompMask            = 0xFF00    /*mask for compression IDs*/
};

/*sound data types*/
enum {
   squareWaveSynth         = 1,        /*square-wave data*/
   waveTableSynth          = 3,        /*wave-table data*/
   sampledSynth            = 5         /*sampled-sound data*/
};

/*sound command numbers*/
enum {
   nullCmd                 = 0,        /*do nothing*/
   quietCmd                = 3,        /*stop a sound that is playing*/
   flushCmd                = 4,        /*flush a sound channel*/
   reInitCmd               = 5,        /*reinitialize a sound channel*/
   waitCmd                 = 10,       /*suspend processing in a channel*/
   pauseCmd                = 11,       /*pause processing in a channel*/
   resumeCmd               = 12,       /*resume processing in a channel*/
   callBackCmd             = 13,       /*execute a callback procedure*/
   syncCmd                 = 14,       /*synchronize channels*/
   availableCmd            = 24,       /*see if initialization options */
                                       /* are supported*/
   versionCmd              = 25,       /*determine version*/
   totalLoadCmd            = 26,       /*report total CPU load*/
   loadCmd                 = 27,       /*report CPU load for a new channel*/
   freqDurationCmd         = 40,       /*play a note for a duration*/
   restCmd                 = 41,       /*rest a channel for a duration*/
   freqCmd                 = 42,       /*change the pitch of a sound*/
   ampCmd                  = 43,       /*change the amplitude of a sound*/
```

```
   timbreCmd               = 44,       /*change the timbre of a sound*/
   getAmpCmd               = 45,       /*get the amplitude of a sound*/
   volumeCmd               = 46,       /*set volume*/
   getVolumeCmd            = 47,       /*get volume*/
   waveTableCmd            = 60,       /*install a wave table as a voice*/
   soundCmd                = 80,       /*install a sampled sound as a voice*/
   bufferCmd               = 81,       /*play a sampled sound*/
   rateCmd                 = 82,       /*set the pitch of a sampled sound*/
   getRateCmd              = 85        /*get the pitch of a sampled sound*/
};

/*sampled sound header encoding options*/
enum {
   stdSH                   = 0x00,     /*standard sound header*/
   extSH                   = 0xFF,     /*extended sound header*/
   cmpSH                   = 0xFE      /*compressed sound header*/
};

/*size of data structures*/
enum {
   stdQLength              = 128       /*default size of sound channel*/
};

/*sound resource formats*/
enum {
   firstSoundFormat        = 0x0001,   /*format 1 'snd ' resource*/
   secondSoundFormat       = 0x0002    /*format 2 'snd ' resource*/
};

/*sound command mask*/
enum {
   dataOffsetFlag          = 0x8000    /*sound command data offset bit*/
};

/*system beep states*/
enum {
   sysBeepDisable          = 0x0000,   /*system alert sound disabled*/
   sysBeepEnable           = 0x0001    /*system alert sound enabled*/
};

/*values for the unitType field in AudioSelection*/
enum {
   unitTypeSeconds         = 0x0000,   /*seconds*/
   unitTypeNoSelection     = 0xFFFF    /*no selection*/
};
```

```
/*double buffer status flags*/
enum {
   dbBufferReady           = 0x00000001,  /*double buffer is filled*/
   dbLastBuffer            = 0x00000004   /*last double buffer to play*/
};

/*values for the compressionID field of CmpSoundHeader*/
enum {
   variableCompression     = -2,       /*variable-ratio compression*/
   fixedCompression        = -1,       /*fixed-ratio compression*/
   notCompressed           = 0,        /*noncompressed samples*/
   threeToOne              = 3,        /*3:1 compressed samples*/
   sixToOne                = 4         /*6:1 compressed samples*/
};

/*values for the packetSize field of CmpSoundHeader*/
enum {
   sixToOnePacketSize      = 8,        /*packet size in bits for 6:1*/
   threeToOnePacketSize    = 16        /*packet size in bits for 3:1*/
};

/*compression names and types*/
#define NoneName              "\pnot compressed"
#define ACE2to1Name           "\pACE 2-to-1"
#define ACE8to3Name           "\pACE 8-to-3"
#define MACE3to1Name          "\pMACE 3-to-1"
#define MACE6to1Name          "\pMACE 6-to-1"
#define NoneType              'NONE'
#define ACE2Type              'ACE2'
#define ACE8Type              'ACE8'
#define MACE3Type             'MAC3'
#define MACE6Type             'MAC6'

/*IDs for AIFF and AIFF-C files*/
#define AIFFID                'AIFF'   /*AIFF file*/
#define AIFCID                'AIFC'   /*AIFF-C file*/

/*IDs for AIFF and AIFF-C file chunks*/
#define FORMID                'FORM'   /*ID for Form Chunk*/
#define FormatVersionID       'FVER'   /*ID for Format Version Chunk*/
#define CommonID              'COMM'   /*ID for Common Chunk*/
#define SoundDataID           'SSND'   /*ID for Sound Data Chunk*/
#define MarkerID              'MARK'   /*ID for Marker Chunk*/
#define InstrumentID          'INST'   /*ID for Instrument Chunk*/
```

```
#define MIDIDataID            'MIDI'   /*ID for MIDI Data Chunk*/
#define AudioRecordingID      'AESD'   /*ID for Recording Chunk*/
#define ApplicationSpecificID 'APPL'   /*ID for Application Chunk*/
#define CommentID             'COMT'   /*ID for Comment Chunk*/
#define NameID                'NAME'   /*ID for Name Chunk*/
#define AuthorID              'AUTH'   /*ID for Author Chunk*/
#define CopyrightID           '(c) '   /*ID for Copyright Chunk*/
#define AnnotationID          'ANNO'   /*ID for Annotation Chunk*/

/*version of AIFC format specification*/
#define AIFCVersion1          0xA2805140
                                       /*date of version creation*/

/*MIDI note value for middle C*/
enum {
   kMiddleC               = 60
};

/*ratio between frequencies of MIDI note values*/
#define twelfthRootTwo        1.05946309434

/*standard sampling rates*/
#define rate44khz          0xAC440000     /*44100.00000 in fixed-point*/
#define rate22khz          0x56EE8BA3     /*22254.54545 in fixed-point*/
#define rate22050hz        0x56220000     /*22050.00000 in fixed-point*/
#define rate11khz          0x2B7745D1     /*11127.27273 in fixed-point*/
#define rate11025hz        0x2B110000     /*11025.00000 in fixed-point*/

/*constant for synth parameter of SndNewChannel*/
enum {
   kUseOptionalOutputDevice      = -1
};

/*volumes*/
enum {
   kFullVolume                   = 0x0100,
   kNoVolume                     = 0
};

/*development stages*/
enum {
   developStage         = 0x20,        /*prealpha release*/
   alphaStage           = 0x40,        /*alpha release*/
```

```
   betaStage            = 0x60,        /*beta release*/
   finalStage           = 0x80         /*final release*/
};

/*sizes of data buffers*/
enum {
   stateBlockSize       = 64,          /*size of state block buffer*/
   leftOverBlockSize    = 32           /*size of leftover block buffer*/
};
```


Data Types

Unsigned Fixed-Point Numbers

```
typedef unsigned long UnsignedFixed;   /*unsigned fixed-point number*/
```

Times

```
typedef long Time;                     /*in half-milliseconds*/
```

Sound Command Record

```
struct SndCommand {
   unsigned short       cmd;           /*command number*/
   short                param1;        /*first parameter*/
   long                 param2;        /*second parameter*/
};
typedef struct SndCommand SndCommand;
```

Audio Selection Record

```
struct AudioSelection {
   long                 unitType;      /*type of time unit*/
   Fixed                selStart;      /*starting point of selection*/
   Fixed                selEnd;        /*ending point of selection/*
};
typedef struct AudioSelection AudioSelection;
typedef AudioSelection *AudioSelectionPtr;
```

Sound Channel Status Record

```
struct SCStatus {
   Fixed               scStartTime;   /*starting time for play from disk*/
   Fixed               scEndTime;     /*ending time for play from disk*/
   Fixed               scCurrentTime; /*current time for play from disk*/
   Boolean             scChannelBusy; /*TRUE if channel is processing cmds*/
   Boolean             scChannelDisposed;
                                      /*reserved*/
   Boolean             scChannelPaused;
                                      /*TRUE if play from disk is paused*/
   Boolean             scUnused;      /*unused*/
   unsigned long       scChannelAttributes;
                                      /*attributes of this channel*/
   long                scCPULoad;     /*CPU load for this channel*/
};
typedef struct SCStatus SCStatus;
typedef SCStatus *SCStatusPtr;
```

Sound Manager Status Record

```
struct SMStatus {
   short               smMaxCPULoad;  /*maximum load on all channels*/
   short               smNumChannels; /*number of allocated channels*/
   short               smCurCPULoad;  /*current load on all channels*/
};
typedef struct SMStatus SMStatus;
typedef SMStatus *SMStatusPtr;
```

Sound Channel Record

```
struct SndChannel {
   struct SndChannel   *nextChan;     /*pointer to next channel*/
   Ptr                 firstMod;      /*used internally*/
   SndCallBackProcPtr  callBack;      /*pointer to callback procedure*/
   long                userInfo;      /*free for application's use*/
   long                wait;          /*used internally*/
   SndCommand          cmdInProgress; /*used internally*/
   short               flags;         /*used internally*/
   short               qLength;       /*used internally*/
   short               qHead;         /*used internally*/
   short               qTail;         /*used internally*/
   SndCommand          queue[stdQLength];
```

```
};
typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

Sound Header Record

```
struct SoundHeader {
   Ptr                 samplePtr;     /*if NIL, samples in sampleArea*/
   unsigned long       length;        /*number of samples in array*/
   Fixed               sampleRate;    /*sample rate for this sound*/
   unsigned long       loopStart;     /*loop point beginning*/
   unsigned long       loopEnd;       /*loop point ending*/
   unsigned char       encode;        /*sample's encoding option*/
   unsigned char       baseFrequency; /*base frequency of sample*/
   unsigned char       sampleArea[1];
};
typedef struct SoundHeader SoundHeader;
typedef SoundHeader *SoundHeaderPtr;
```

Extended Sound Header Record

```
struct ExtSoundHeader {
   Ptr                 samplePtr;     /*if NIL, samples in sampleArea*/
   unsigned long       numChannels;   /*number of channels in sample*/
   Fixed               sampleRate;    /*rate of original sample*/
   unsigned long       loopStart;     /*loop point beginning*/
   unsigned long       loopEnd;       /*loop point ending*/
   unsigned char       encode;        /*sample's encoding option*/
   unsigned char       baseFrequency; /*base frequency of sample*/
   unsigned long       numFrames;     /*total number of frames*/
   extended80          AIFFSampleRate;/*rate of original sample*/
   Ptr                 markerChunk;   /*reserved*/
   Ptr                 instrumentChunks;
                                      /*pointer to instrument info*/
   Ptr                 AESRecording;  /*pointer to audio info*/
   unsigned short      sampleSize;    /*number of bits per sample*/
   unsigned short      futureUse1;    /*reserved*/
   unsigned long       futureUse2;    /*reserved*/
   unsigned long       futureUse3;    /*reserved*/
   unsigned long       futureUse4;    /*reserved*/
   unsigned char       sampleArea[1];
};
typedef struct ExtSoundHeader ExtSoundHeader;
typedef ExtSoundHeader *ExtSoundHeaderPtr;
```

Compressed Sound Header Record

```
struct CmpSoundHeader {
   Ptr                  samplePtr;     /*if NIL, samples in sampleArea*/
   unsigned long        numChannels;   /*number of channels in sample*/
   Fixed                sampleRate;    /*rate of original sample*/
   unsigned long        loopStart;     /*loop point beginning*/
   unsigned long        loopEnd;       /*loop point ending*/
   unsigned char        encode;        /*sample's encoding option*/
   unsigned char        baseFrequency; /*base frequency of original sample*/
   unsigned long        numFrames;     /*length of sample in frames*/
   extended80           AIFFSampleRate;/*rate of original sample*/
   Ptr                  markerChunk;   /*reserved*/
   OSType               format;        /*data format type*/
   unsigned long        futureUse2;    /*reserved*/
   StateBlockPtr        stateVars;     /*pointer to StateBlock*/
   LeftOverBlockPtr     leftOverSamples;
                                       /*pointer to LeftOverBlock*/
   unsigned short       compressionID; /*ID of compression algorithm*/
   unsigned short       packetSize;    /*number of bits per packet*/
   unsigned short       snthID;        /*unused*/
   unsigned short       sampleSize;    /*bits in each sample point*/
   unsigned char        sampleArea[1];
};
typedef struct CmpSoundHeader CmpSoundHeader;
typedef CmpSoundHeader *CmpSoundHeaderPtr;
```

Sound Double Buffer Header Record

```
struct SndDoubleBufferHeader {
   short                dbhNumChannels;/*number of sound channels*/
   short                dbhSampleSize; /*sample size, if noncompressed*/
   short                dbhCompressionID;
                                       /*ID of compression algorithm*/
   short                dbhPacketSize; /*number of bits per packet*/
   Fixed                dbhSampleRate; /*sample rate*/
   SndDoubleBufferPtr   dbhBufferPtr[2];
                                       /*pointers to SndDoubleBuffer*/
   SndDoubleBackProcPtr dbhDoubleBack; /*pointer to doubleback procedure*/
};
typedef struct SndDoubleBufferHeader SndDoubleBufferHeader;
typedef SndDoubleBufferHeader *SndDoubleBufferHeaderPtr;
```

```
struct SndDoubleBufferHeader2 {
   short                dbhNumChannels;/*number of sound channels*/
   short                dbhSampleSize; /*sample size, if noncompressed*/
   short                dbhCompressionID;
                                       /*ID of compression algorithm*/
   short                dbhPacketSize; /*number of bits per packet*/
   Fixed                dbhSampleRate; /*sample rate*/
   SndDoubleBufferPtr   dbhBufferPtr[2];
                                       /*pointers to SndDoubleBuffer*/
   SndDoubleBackProcPtr dbhDoubleBack; /*pointer to doubleback procedure*/
   OSType               dbhFormat;     /*signature of codec*/
};
typedef struct SndDoubleBufferHeader2 SndDoubleBufferHeader2;
typedef SndDoubleBufferHeader2 *SndDoubleBufferHeaderPtr2;
```

Sound Double Buffer Record

```
struct SndDoubleBuffer {
   long                 dbNumFrames;   /*number of frames in buffer*/
   long                 dbFlags;       /*buffer status flags*/
   long                 dbUserInfo[2]; /*for application's use*/
   char                 dbSoundData[1];/*array of data*/
};
typedef struct SndDoubleBuffer SndDoubleBuffer;
typedef SndDoubleBuffer *SndDoubleBufferPtr;
```

Chunk Headers

```
typedef unsigned long ID;              /*chunk ID type*/

struct ChunkHeader {
   ID                   ckID;          /*chunk type ID*/
   long                 ckSize;        /*number of bytes of data*/
};
typedef struct ChunkHeader ChunkHeader;
```

Form Chunk

```
struct ContainerChunk {
   ID                   ckID;          /*'FORM'*/
   long                 ckSize;        /*number of bytes of data*/
   ID                   formType;      /*type of file*/
};
typedef struct ContainerChunk ContainerChunk;
```

Format Version Chunk

```
struct FormatVersionChunk {
   ID                   ckID;          /*'FVER'*/
   long                 ckSize;        /*4 bytes*/
   unsigned long        timestamp;     /*date of format version*/
};
typedef struct FormatVersionChunk FormatVersionChunk;
```

Common Chunk

```
struct CommonChunk {
   ID                   ckID;          /*'COMM'*/
   long                 ckSize;        /*18 bytes*/
   short                numChannels;   /*number of channels*/
   unsigned long        numSampleFrames;
                                       /*number of sample frames*/
   short                sampleSize;    /*number of bits per sample*/
   extended80           sampleRate;    /*number of frames per second*/
};

typedef struct CommonChunk CommonChunk;
```

Extended Common Chunk

```
struct ExtCommonChunk {
   ID                   ckID;          /*'COMM'*/
   long                 ckSize;        /*22 bytes + compression name*/
   short                numChannels;   /*number of channels*/
   unsigned long        numSampleFrames;
                                       /*number of sample frames*/
   short                sampleSize;    /*number of bits per sample*/
   extended80           sampleRate;    /*number of frames per second*/
   ID                   compressionType;
                                       /*compression type ID*/
   char                 compressionName[1];
                                       /*compression type name*/
};
typedef struct ExtCommonChunk ExtCommonChunk;
```

Sound Data Chunk

```
struct SoundDataChunk {
   ID                   ckID;          /*'SSND'*/
   long                 ckSize;        /*size of chunk data*/
   unsigned long        offset;        /*offset to sound data*/
   unsigned long        blockSize;     /*size of alignment blocks*/
};
typedef struct SoundDataChunk SoundDataChunk;
```

Version Record

```
struct NumVersion {
   unsigned char        majorRev;      /*major revision level in BCD*/
   unsigned char        minorAndBugRev;/*minor revision level*/
   unsigned char        stage;         /*development stage*/
   unsigned char        nonRelRev;     /*nonreleased version revision level*/
};
typedef struct NumVersion NumVersion;
```

Leftover Block

```
struct LeftOverBlock {
   unsigned long        count;
   char                 sampleArea[leftOverBlockSize];
};
typedef struct LeftOverBlock LeftOverBlock;
typedef LeftOverBlock *LeftOverBlockPtr;
```

State Block

```
struct StateBlock {
   short                stateVar[stateBlockSize];
};
typedef struct StateBlock StateBlock;
typedef StateBlock *StateBlockPtr;
```

Procedure Types

```
typedef pascal void (*FilePlayCompletionProcPtr)
                           (SndChannelPtr chan);
typedef pascal void (*SndCallBackProcPtr)
                           (SndChannelPtr chan, SndCommand *cmd);
```

```
typedef pascal void (*SndDoubleBackProcPtr)
                            (SndChannelPtr chan,
                             SndDoubleBufferPtr doubleBufferPtr);
```

Sound Manager Routines

Playing Sound Resources

```
pascal void SysBeep         (short duration);
pascal OSErr SndPlay        (SndChannelPtr chan, Handle sndHdl,
                             Boolean async);
```

Playing From Disk

```
pascal OSErr SndStartFilePlay
                            (SndChannelPtr chan, short fRefNum,
                             short resNum, long bufferSize, void *theBuffer,
                             AudioSelectionPtr theSelection,
                             FilePlayCompletionProcPtr theCompletion,
                             Boolean async);
pascal OSErr SndPauseFilePlay
                            (SndChannelPtr chan);
pascal OSErr SndStopFilePlay
                            (SndChannelPtr chan, Boolean quietNow);
```

Allocating and Releasing Sound Channels

```
pascal OSErr SndNewChannel  (SndChannelPtr *chan, short synth, long init,
                             SndCallBackProcPtr userRoutine);
pascal OSErr SndDisposeChannel
                            (SndChannelPtr chan, Boolean quietNow);
```

Sending Commands to a Sound Channel

```
pascal OSErr SndDoCommand   (SndChannelPtr chan, const SndCommand *cmd,
                             Boolean noWait);
pascal OSErr SndDoImmediate
                            (SndChannelPtr chan, const SndCommand *cmd);
```

Obtaining Information

```
pascal NumVersion SndSoundManagerVersion
                            (void);
pascal NumVersion MACEVersion
                            (void);
```

```
pascal OSErr SndControl     (short id, SndCommand *cmd);
pascal OSErr SndChannelStatus
                            (SndChannelPtr chan, short theLength,
                             SCStatusPtr theStatus);
pascal OSErr SndManagerStatus
                            (short theLength, SMStatusPtr theStatus);
pascal void SndGetSysBeepState
                            (short *sysBeepState);
pascal OSErr SndSetSysBeepState
                            (short sysBeepState);
pascal OSErr GetSoundHeaderOffset
                            (Handle sndHandle, long *offset);
```

Controlling Volume Levels

```
pascal OSErr GetSysBeepVolume
                            (long *level);
pascal OSErr SetSysBeepVolume
                            (long level);
pascal OSErr GetDefaultOutputVolume
                            (long *level);
pascal OSErr SetDefaultOutputVolume
                            (long level);
```

Compressing and Expanding Audio Data

```
pascal void Comp3to1        (const void *inBuffer, void *outBuffer,
                             unsigned long cnt, const void *inState,
                             void *outState, unsigned long numChannels,
                             unsigned long whichChannel);
pascal void Comp6to1        (const void *inBuffer, void *outBuffer,
                             unsigned long cnt, const void *inState,
                             void *outState, unsigned long numChannels,
                             unsigned long whichChannel);
pascal void Exp1to3         (const void *inBuffer, void *outBuffer,
                             unsigned long cnt, const void *inState,
                             void *outState, unsigned long numChannels,
                             unsigned long whichChannel);
pascal void Exp1to6         (const void *inBuffer, void *outBuffer,
                             unsigned long cnt, const void *inState,
                             void *outState, unsigned long numChannels,
                             unsigned long whichChannel);
```

Managing Double Buffers

```
pascal OSErr SndPlayDoubleBuffer
                            (SndChannelPtr chan,
                             SndDoubleBufferHeaderPtr theParams);
```

Performing Unsigned Fixed-Point Arithmetic

```
pascal UnsignedFixed UnsignedFixMulDiv
                            (UnsignedFixed value, UnsignedFixed multiplier,
                             UnsignedFixed divisor);
```

Linking Modifiers to Sound Channels

```
pascal OSErr SndAddModifier
                            (SndChannelPtr chan, Ptr modifier, short id,
                             long init);
```

Application-Defined Routines

```
pascal void MyFilePlayCompletionRoutine
                            (SndChannelPtr chan);
pascal void MyCallback      (SndChannelPtr chan, SndCommand *cmd);
pascal void MyDoubleBackProc
                            (SndChannelPtr chan,
                             SndDoubleBufferPtr doubleBufferPtr);
```

Assembly-Language Summary

Data Structures

SndCommand Data Structure

0	`cmd`	word	command number
2	`param1`	word	first parameter
4	`param2`	long	second parameter

AudioSelection Data Structure

0	`unitType`	long	type of time unit
4	`selStart`	4 bytes	starting point of selection (Fixed)
8	`selEnd`	4 bytes	ending point of selection (Fixed)

SCStatus Data Structure

0	scStartTime	4 bytes	starting time for play from disk (Fixed)
4	scEndTime	4 bytes	ending time for play from disk (Fixed)
8	scCurrentTime	4 bytes	current time for play from disk (Fixed)
12	scChannelBusy	byte	channel playing sampled sound flag
13	scChannelDisposed	byte	reserved
14	scChannelPaused	byte	play from disk is paused flag
15	scUnused	byte	unused
16	scChannelAttributes	long	attributes of channel
20	scCPULoad	long	CPU load for channel

SMStatus Data Structure

0	smMaxCPULoad	word	maximum load on all channels
2	smNumChannels	word	number of allocated channels
4	smCurCPULoad	word	current load on all channels

SndChannel Data Structure

0	nextChan	long	pointer to next channel
4	firstMod	long	used internally
8	callBack	long	pointer to callback procedure
12	userInfo	long	free for application's use
16	wait	long	used internally
20	cmdInProgress	8 bytes	used internally
28	flags	word	used internally
30	qLength	word	used internally
32	qHead	word	used internally
34	qTail	word	used internally
36	queue	variable	queue of sound commands

SoundHeader Data Structure

0	samplePtr	long	pointer to samples (or NIL if samples follow data structure)
4	length	long	number of samples in array
8	sampleRate	4 bytes	sample rate (Fixed)
12	loopStart	long	loop point beginning
16	loopEnd	long	loop point ending
20	encode	byte	sample's encoding option
21	baseFrequency	byte	base frequency of sample
22	sampleArea	variable	sampled-sound data

ExtSoundHeader Data Structure

0	samplePtr	long	pointer to samples (or NIL if samples follow data structure)
4	numChannels	long	number of channels in sample
8	sampleRate	4 bytes	sample rate (Fixed)
12	loopStart	long	loop point beginning
16	loopEnd	long	loop point ending

20	encode	byte	sample's encoding option
21	baseFrequency	byte	base frequency of sample
22	numFrames	long	total number of frames
26	AIFFSampleRate	10 bytes	rate of original sample (Extended80)
36	markerChunk	long	reserved
40	instrumentChunks	long	pointer to instrument info
44	AESRecording	long	pointer to audio info
48	sampleSize	word	number of bits per sample
50	futureUse1	word	reserved
52	futureUse2	long	reserved
56	futureUse3	long	reserved
60	futureUse4	long	reserved
64	sampleArea	variable	sampled-sound data

CmpSoundHeader Data Structure

0	samplePtr	long	pointer to samples (or NIL if samples follow data structure)
4	numChannels	long	number of channels in sample
8	sampleRate	4 bytes	sample rate (Fixed)
12	loopStart	long	loop point beginning
16	loopEnd	long	loop point ending
20	encode	byte	sample's encoding option
21	baseFrequency	byte	base frequency of original sample
22	numFrames	long	length of sample in frames
26	AIFFSampleRate	10 bytes	rate of original sample (Extended80)
36	markerChunk	long	reserved
40	format	OSType	data format type
44	futureUse2	long	reserved
48	stateVars	long	pointer to StateBlock
52	leftOverSamples	long	pointer to LeftOverBlock
56	compressionID	word	ID of compression algorithm
58	packetSize	word	number of bits per packet
60	snthID	word	unused
62	sampleSize	word	bits in each sample point
64	sampleArea	variable	compressed sound data

SndDoubleBufferHeader Data Structure

0	dbhNumChannels	word	number of sound channels
2	dbhSampleSize	word	sample size, if noncompressed
4	dbhCompressionID	word	ID of compression algorithm
6	dbhPacketSize	word	number of bits per packet
8	dbhSampleRate	4 bytes	sample rate (Fixed)
12	dbhBufferPtr	2 longs	pointers to SndDoubleBuffer data structures
20	dbhDoubleBack	long	pointer to doubleback procedure

SndDoubleBuffer Data Structure

0	dbNumFrames	long	number of frames in buffer
4	dbFlags	long	buffer status flags
8	dbUserInfo	2 longs	for application's use
16	dbSoundData	variable	array of data

ChunkHeader Data Structure

0	ckID	long	chunk type ID
4	ckSize	long	number of bytes of data

ContainerChunk Data Structure

0	ckID	long	chunk type ID ('FORM')
4	ckSize	long	number of bytes of data
8	formType	long	type of file

FormatVersionChunk Data Structure

0	ckID	long	chunk type ID ('FVER')
4	ckSize	long	number of bytes of data (4)
8	timestamp	long	date of format version

CommonChunk Data Structure

0	ckID	long	chunk type ID ('COMM')
4	ckSize	long	number of bytes of data (18)
8	numChannels	word	number of channels
10	numSampleFrames	long	number of sample frames
14	sampleSize	word	number of bits per sample
16	sampleRate	10 bytes	number of frames per second (Extended80)

ExtCommonChunk Data Structure

0	ckID	long	chunk type ID ('COMM')
4	ckSize	long	number of bytes of data (22 + length of compression name)
8	numChannels	word	number of channels
10	numSampleFrames	long	number of sample frames
14	sampleSize	word	number of bits per sample
16	sampleRate	10 bytes	number of frames per second (Extended80)
26	compressionType	long	compression type ID
30	compressionName	variable	compression type name

SoundDataChunk

0	ckID	long	chunk type ID ('SSND')
4	ckSize	long	number of bytes of data
8	offset	long	offset to sound data
12	blockSize	long	size of alignment blocks

Trap Macros

Trap Macro Requiring Routine Selectors

_SoundDispatch

Selector	Routine
$00000010	MACEVersion
$00040010	Comp3to1
$00080010	Exp1to3
$000C0008	SndSoundManagerVersion
$000C0010	Comp6to1
$00100008	SndChannelStatus
$00100010	Exp1to6
$00140008	SndManagerStatus
$00180008	SndGetSysBeepState
$001C0008	SndSetSysBeepState
$00200008	SndPlayDoubleBuffer
$02040008	SndPauseFilePlay
$02240024	GetSysBeepVolume
$02280024	SetSysBeepVolume
$022C0024	GetDefaultOutputVolume
$02300024	SetDefaultOutputVolume
$03080008	SndStopFilePlay
$0D000008	SndStartFilePlay
$04040024	GetSoundHeaderOffset

Result Codes

noErr	0	No error
paramErr	–50	A parameter is incorrect
noHardwareErr	–200	Required sound hardware not available
notEnoughHardwareErr	–201	Insufficient hardware available
queueFull	–203	No room in the queue
resProblem	–204	Problem loading the resource
badChannel	–205	Channel is corrupt or unusable
badFormat	–206	Resource is corrupt or unusable
notEnoughBufferSpace	–207	Insufficient memory available
badFileFormat	–208	File is corrupt or unusable, or not AIFF or AIFF-C
channelBusy	–209	Channel is busy
buffersTooSmall	–210	Buffer is too small

`channelNotBusy`	–211	Channel not currently used
`noMoreRealTime`	–212	Not enough CPU time available
`siInvalidCompression`	–223	Invalid compression type