Mac OS Sound

Including Sound Manager 3.3

Apple, the Apple logo, APDA, HyperCard, LaserWriter, Macintosh, Macintosh Quadra, MPW, and PowerBook are trademarks of Apple Computer, Inc., registered in the United States and other countries.

AppleDesign, AudioVision, Finder, MacinTalk, QuickDraw, and QuickTime are trademarks of Apple Computer, Inc.

Adobe Illustrator, Adobe Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

Classic® is a registered trademark licensed to Apple Computer, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

Internet is a trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

NuBus™ is a trademark of Texas Instruments.

Optrotech is a trademark of Orbotech Corporation.

Sony™ is a trademark of Sony Corporation, registered in the U.S. and other countries.

Windows is a trademark of Microsoft Corporation.

Simultaneously published in the United States and Canada.

Contents

Chapter 5   Sound Manager 3.3 Features   315

Chapter 6   Sound Manager 3.3 Reference   335

Glossary   351

Index   359

Figures, Tables, and Listings

10

About This Book

This book, *Mac OS Sound,* describes the parts of the Macintosh system software that allow you to manage sounds. It describes the services provided by the three principal sound-related system software managers (the Sound Manager and the Sound Input Manager) and shows in detail how your application can record and play back sounds, compress and expand audio data, convert text to speech, and perform other similar operations.

If you are not yet experienced with playing or recording sounds on Macintosh computers, you should begin with the chapter "Introduction to Sound on the Macintosh." That chapter describes the services provided by the system software and shows how to use the most basic sound-related capabilities of Macintosh computers. It provides complete source code examples illustrating how to record sounds into resources and files, how to play sounds stored in resources and files, and how to convert written text into spoken words. It's possible that this introductory chapter contains all the information you need to successfully integrate sound into your application.

Once you are familiar with basic sound recording and production on Macintosh computers, you might want to read other chapters in this book. The chapter "Sound Manager" provides complete information about sound output. It shows how to control sound production at a very low level, how to produce sound asynchronously (that is, while other operations in the computer take place), and how to compress and expand audio data. This chapter also provides complete details about the structure of the two main sound storage formats, sound resources and sound files.

If you need more control over the sound recording process than is offered by the basic recording functions described in the chapter "Introduction to Sound on the Macintosh," you need to read the chapter "Sound Input Manager." That chapter shows how to record sound without displaying the sound recording dialog box or to interact directly with a sound input device driver.

The chapter "Sound Components" describes how to write sound components. The Sound Manager uses sound components to manipulate audio data or to communicate with sound output devices. You need to read this chapter only if you are developing a new sound output device or want to use a custom audio data compression and expansion scheme.

Finally, the chapters "Sound Manager 3.3: Features" and "Sound Manager 3.3: Reference" bring you up to date. They contain information about changes and improvements to the Sound Manager from release 3.0 to the latest version, release 3.3. An important update is that the Mac OS Sound Manager code is now compatible with Windows 95 and Windows NT.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Sound Input Manager" contains these sections:

■ "About the Sound Input Manager." This section provides an overview of the features provided by the Sound Input Manager.

■ "Using the Sound Input Manager." This section describes the tasks you can accomplish using the Sound Input Manager. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information.

■ "Sound Input Manager Reference." This section provides a complete reference for the Sound Input Manager by describing the constants, data structures, routines, and resources it uses. Each routine description also follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.

Conventions Used in This Book

This book uses special conventions to present certain types of information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in `Courier` (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 1-21.) ◆

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-24.) ▲

▲ **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 2-79.) ▲

Assembly-Language Information

This book provides information about the registers for specific routines in this format:

Registers on entry

A0   Contents of register A0 on entry

Registers on exit

D0   Contents of register D0 on exit

In addition, this book presents information about the fields of a parameter block in this format:

Parameter block

↔	`inAndOut`	`Integer`	Input/output parameter.
←	`output1`	`Ptr`	Output parameter.
→	`input1`	`Ptr`	Input parameter.

The arrow in the far left column indicates whether the field is an input parameter, output parameter, or both. You must supply values for all input parameters and input/output parameters. The routine returns values in output parameters and input/output parameters.

The second column shows the field name as defined in the interface files; the third column indicates the data type of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion that follows the parameter block or the description of the parameter block in the reference section of the chapter.

Development Environment

The system software routines described in this book are available using C interfaces. How you access these routines depends on the development environment you are using. When showing system software routines, this book sometimes uses Pascal interfaces. However, the chapters "Sound

Components," "Sound Manager 3.3: Features," and "Sound Manager 3.3: Reference"use C interfaces.

All code listings in this book are shown in Pascal or C. They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer, Inc. does not intend for you to use these code samples in your application.

This book occasionally illustrates concepts by referring to a sample application called SurfWriter. This application is not an actual product of Apple Computer, Inc. This book also uses the names SurfBoard and WaveMaker to refer to sample sound output and input devices. These devices are not actual products of Apple Computer, Inc.

Introduction to Sound on the Macintosh

This chapter provides an introduction to managing sound on Macintosh computers. It's intended to help you quickly get started integrating sound into your application. This chapter introduces the concepts described in detail throughout the rest of this book and provides source code examples that show you how to use the most basic sound-related capabilities of Macintosh computers. These examples use the Sound Manager to play sounds and the Sound Input Manager to record sounds.

Even if your application is not specifically concerned with creating or playing sounds, you can often improve your application at very little programming expense by using these system software services to integrate sound or speech into its user interface. For example, you might use the techniques described in this chapter to

■ play a sound to alert the user that a lengthy spreadsheet calculation is completed

■ provide voice annotations for a word-processing document

■ read aloud the text string that is displayed in a dialog box

If you want to use sound in these simple ways, this chapter will probably provide all the information you need. The Sound Manager and Sound Input Manager provide high-level routines that make it very easy to play or record sounds without knowing very much about how sounds are stored or produced electronically.

If, on the other hand, you are writing an application that is primarily concerned with sound, you should read this chapter and some of the remaining chapters in this book. You also need to read those chapters if you want to play computer-generated tones without using sound resources or sound files, play sounds asynchronously, play sounds at different pitches, record sounds without using the standard sound recording interface, or customize the quality of speech output to make it easier to understand.

To benefit most from this chapter, you should already be familiar with simple resource and file management, discussed in the chapters "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* and "Introduction to File Management" in *Inside Macintosh: Files.*

In particular, this chapter does not explain how to open or close resource or data files, although it does provide source code examples that demonstrate how to play a sound from, or record a sound to, a resource or data file that is already open.

This chapter begins with an overview of sound on Macintosh computers. It describes the audio capabilities available on all Macintosh computers and some of the capabilities achievable by adding additional hardware and software to Macintosh computers. Then this chapter describes how you can use the available system software routines to

- play the system alert sound
- play sounds stored as resources
- play sampled sounds stored in sound files
- determine whether a particular Macintosh computer is capable of recording sounds
- record sounds into resources
- record sounds into sound files
- convert text strings into spoken words

For your convenience, this chapter also includes a reference section containing complete descriptions of the routines used to perform these tasks, and both Pascal and C language summaries. All of the routines in the reference section of this chapter are also in the reference sections of the chapter that describes the manager they are part of.

About Sound on Macintosh Computers

The Macintosh hardware and system software provide a standard and extensible set of capabilities for producing and recording sounds. No matter what kind of application you are developing, you can use these capabilities to enrich your application, often at very little programming expense. For example, you might allow users to attach voice annotations to documents or to other collections of data. Or, you might play a certain sound to signal that some operation has completed.

This section provides a general overview of the sound input and output capabilities available on Macintosh computers. It defines some of the concepts used throughout this book and describes how sounds can be stored by your application. This section also describes the standard ways of representing sounds in the Macintosh graphical user interface.

Sound Capabilities

The Macintosh family of computers provides sound input and output capabilities that far exceed the capabilities of most other personal computers. The principal reason for this is that the hardware and software aspects of creating or recording sounds are more tightly integrated with one another than they are on other personal computers.

Figure 1-1 illustrates the basic audio hardware and the sound-related system software that are now standard on all Macintosh computers.

Figure 1-1   Basic sound capabilities on Macintosh computers
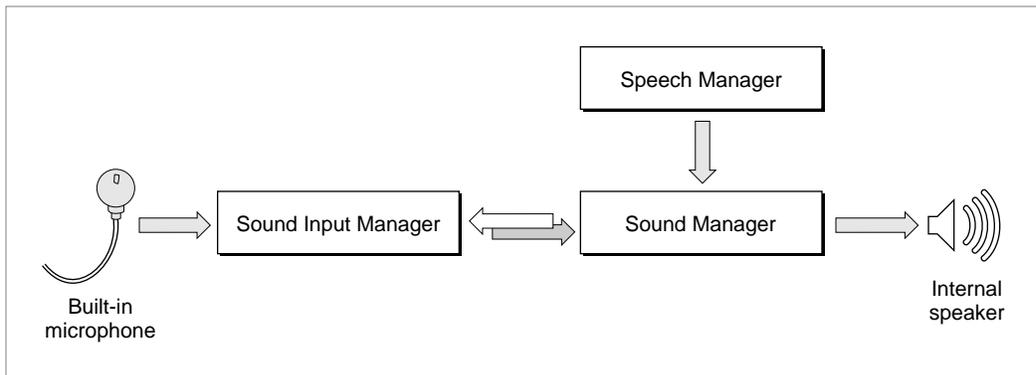
The audio hardware includes an internal speaker (for producing sounds), a microphone (for recording sounds), and one or more integrated circuits that convert digital data to analog signals, or analog signals to digital data. The actual integrated circuits that perform the conversion of digital to analog data (and vice versa) vary among different models of Macintosh computers. What's important is that, together with the available sound-related system software, the basic audio hardware provides a wide range of sound input and output capabilities, including

- playback of digitally recorded (that is, sampled) sounds

- playback of simple sequences of notes or of complex waveforms

- recording of sampled sounds

- conversion of text to spoken words

- mixing and synchronization of multiple channels of sampled sounds

- compression and decompression of sound data to minimize storage space

In general, you'll interact directly with the system software that provides these and other capabilities. The Macintosh sound architecture includes three principal system software services:

- The **Sound Manager** provides the ability to play sounds through the speaker. It also provides an extensive set of tools for manipulating sounds. You can use the Sound Manager to alter virtually any characteristic of a sound, such as its loudness, pitch, timbre, and duration. You can also use the Sound Manager to compress sounds so that they occupy less disk space. The Sound Manager can work with sounds stored in resources or in a file's data fork. It can also play sounds that are generated dynamically (and not necessarily stored on disk).

- The **Sound Input Manager** provides the ability to record sounds through a microphone or other sound input device. It manages the standard sound recording

dialog box (shown in Figure 1-12 on page 31) and can record sounds into resources or into files.

■ The **Speech Manager** provides the ability to convert written text into spoken words. You might use the Speech Manager to read aloud a block of text that for various reasons cannot be sampled (perhaps the amount of text is too large to be recorded and then replayed, or perhaps the text itself is generated dynamically by the user). The Speech Manager allows you to select from among a number of different voices, alter some of the readback characteristics (such as speech, pitch, and volume), and provide custom pronunciation dictionaries.

The basic sound hardware and system software also provide the ability to integrate and synchronize sound production with the display of other types of information, such as video and still images. For example, QuickTime uses the Sound Manager to handle all the sound data in a QuickTime movie.

It's very easy for users to enhance the quality of the sounds they play back or record by substituting different speakers or microphones for the ones built into a Macintosh computer. All current Macintosh computers include a stereo sound output jack that allows users to add high quality speakers (such as the AppleDesign Powered Speakers). A user can also substitute a higher quality microphone for the one supplied with the computer. Figure 1-2 illustrates a slightly better audio configuration than the one shown in Figure 1-1.

Figure 1-2   Enhanced sound capabilities on Macintosh computers
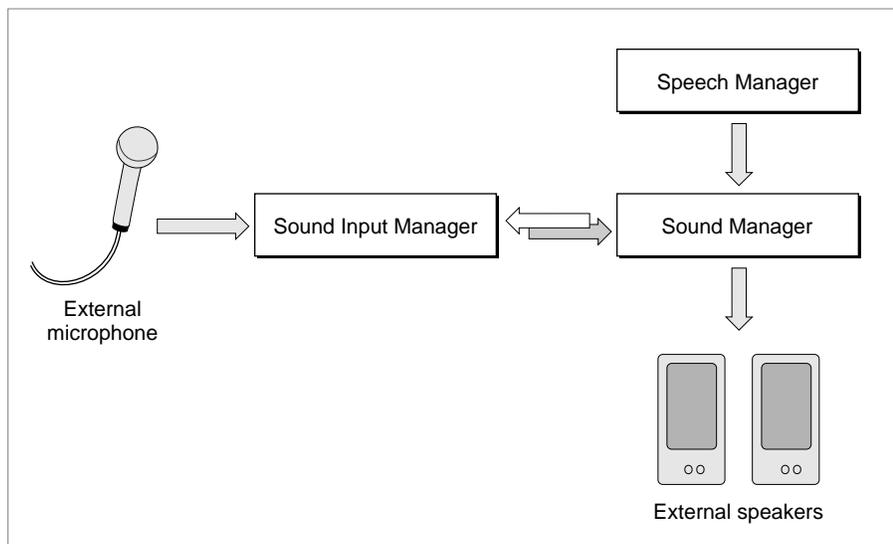
Note that the enhanced sound input and output capabilities shown in Figure 1-2 are provided entirely by the improved hardware. The system software (in particular, the Sound Manager and the Sound Input Manager) can support both the built-in audio hardware and any external hardware connected to the built-in audio jacks.

It's possible to enhance the audio capabilities of a Macintosh computer even further. For example, a user can add a NuBus™ expansion card that contains very high quality digital signal processing (DSP) circuitry, together with sound input or output hardware. These cards typically bypass the standard Macintosh sound circuitry altogether and therefore require additional software (a device driver) to work with the Sound Manager or the Sound Input Manager. The system software is, however, designed to make it easy for developers to add software to drive their sound output or sound input devices.

A user can also enhance the audio capabilities of a Macintosh computer by adding a MIDI interface to one of its serial ports. **MIDI** (the Musical Instrument Digital Interface) is a standard protocol for sending audio data and commands to digital devices. A user can connect any MIDI devices (such as synthesizers, drum machines, or lighting controllers) to a Macintosh computer through the MIDI interface. Apple Computer supplies a software driver, the **MIDI Manager,** to control the flow of MIDI data and commands through the MIDI interface.

Note

The MIDI Manager is not documented in this book. For complete information about the MIDI Manager, contact APDA. ◆

Figure 1-3 illustrates a very high capability sound and music configuration built around a Macintosh computer. This enhanced hardware and system software configuration allows users to run digital sound editing or recording applications and MIDI sequencing applications.

Figure 1-3 High quality sound capabilities on Macintosh computers
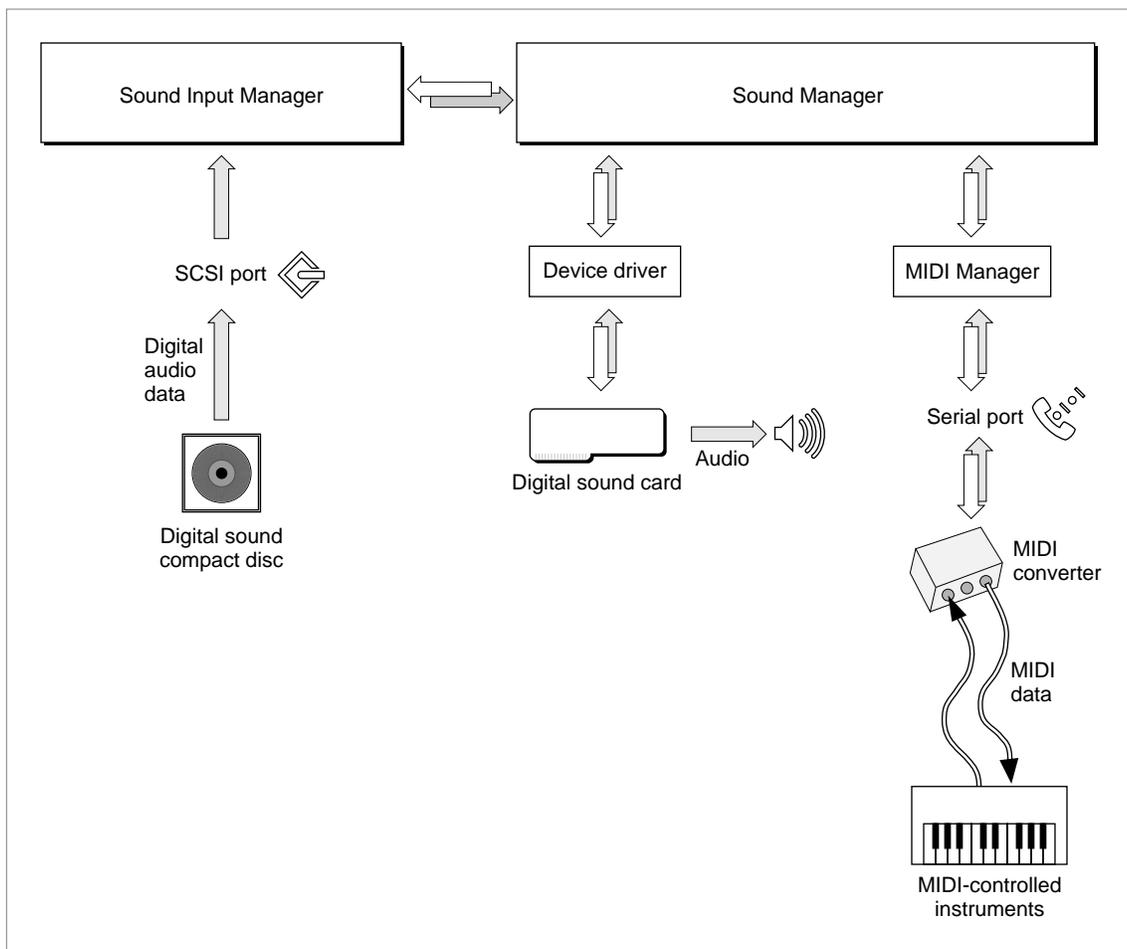
It's possible to enhance the sound environment on a Macintosh computer by adding software alone, for example by adding custom sound **compression/decompression components (codecs).** Apple Computer supplies codecs that can handle 3:1 and 6:1 compression and expansion, which are suitable for most audio requirements. For special purposes, however, it might be advantageous to use other compression and expansion ratios or algorithms. The Sound Manager can use any available codec to handle compression and expansion of audio data.

More generally, the Sound Manager supports arbitrary modifications on sound data using stand-alone code resources known as **sound components.** A sound component can perform one or more signal-processing operations on sound data. For example, the Sound Manager includes sound components for compressing and decompressing sound data (as described in the previous paragraph) and for converting sample rates. Sound components have a standard programming interface and local storage, which allows them to be hooked together in series to perform complex tasks. For instance, to play an 11 kHz compressed sampled sound on a Macintosh II computer, the Sound Manager

needs to expand the compressed data into audio samples, convert the samples from 11 kHz to 22 kHz, mix the samples with any other sounds that are playing, and then send the mixed samples to the available audio hardware (in this case, the Apple Sound Chip). The Sound Manager uses four different sound components to accomplish this task, as shown in Figure 1-4.

Figure 1-4 A sound component chain
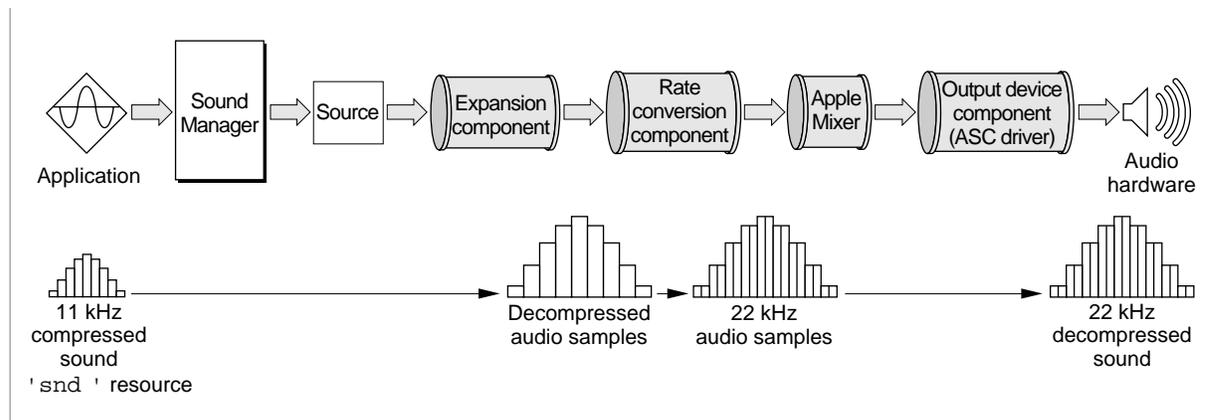
Except for the lowest-level components that communicate directly with hardware (here, the Apple Sound Chip), the components of this chain operate solely on a stream of bytes. This allows Apple and other developers to create sound components that operate independently of the actual sound-producing hardware available on a particular Macintosh computer. This also allows the Sound Manager to modify the component chain used at any time according to the actual capabilities of the output hardware. For example, a digital signal processing card might be able to do rate conversion internally. In that case, the Sound Manager can bypass the rate conversion component and send the 11 kHz samples directly to the DSP card, as shown in Figure 1-5.

Figure 1-5 A sound component chain with a DSP board
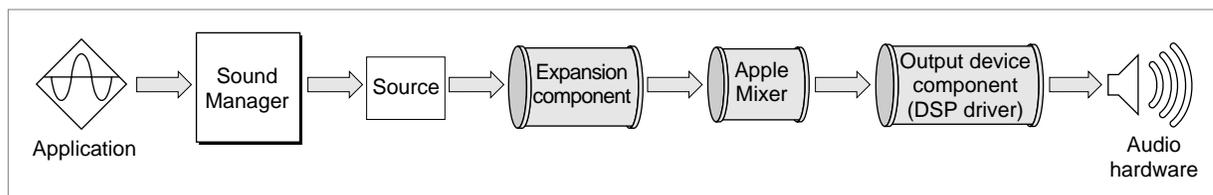
In general, an application that wants to produce a sound is unaware of the sound component chain required to produce that sound on the current sound output device. The Sound Manager keeps track of which sound output device the user has selected and

constructs a component chain suitable for producing the desired quality of sound on that device. As a result, even though the capabilities of the available sound output hardware can vary greatly from one Macintosh computer to another, the Sound Manager ensures that a given chunk of audio data always sounds as good as possible on the available sound hardware. This means that you can use the same code to play sounds, regardless of the actual sound-producing hardware that is available on a particular machine.

The Sound Manager provides sound components for modifying and producing sounds on the built-in audio hardware and on any hardware attached to the sound output jack. The Macintosh sound architecture currently allows you to add sound components for two special purposes: to support alternate compression and decompression algorithms and to support third-party audio hardware. See the chapter "Sound Components" in this book for information on developing codecs and sound output device components.

IMPORTANT

You don't need to know how to develop sound components simply to play or record sounds on Macintosh computers using the available sound output or input devices. ▲

The following sections describe in greater detail the operations of the Sound Manager, the Sound Input Manager, and the Speech Manager. You'll use the Sound Manager to produce sounds, the Sound Input Manager to record sounds, and the Speech Manager to generate speech from text.

Sound Production

A Macintosh computer produces sound when the Sound Manager sends some data through a sound channel to the available audio hardware, usually at the request of an application. The audio hardware is a **digital-to-analog converter (DAC)** that translates digital sound data into analog audio signals. Those signals are then sent to the internal speaker, to a sound output connector (to which the user can connect headphones, external speakers, or sound amplification equipment), or to other sound output hardware.

The DAC in Macintosh Plus and Macintosh SE computers is a Sony sound chip. The Macintosh II, Macintosh Portable, Macintosh PowerBook and Macintosh Quadra families of computers contain two Sony sound chips (to provide stereo output capability) as well as the **Apple Sound Chip (ASC),** a customized chip that provides enhanced audio output characteristics as well as emulation capabilities for the earlier sound hardware.

Some recent models of Macintosh computers contain built-in sound hardware that extends the Apple Sound Chip's features. For example, Macintosh computers with built-in microphones include the **Enhanced Apple Sound Chip (EASC).** Some Macintosh computers contain DSP chips that provide very high-quality sound (16-bit stereo sound, at rates up to 44 kHz). There are also NuBus expansion cards available from third-party developers that provide other audio DAC hardware.

A user can select a sound output device or control characteristics of the selected device through the **Sound Out control panel,** shown in Figure 1-6. The available sound output devices are listed in the center of the panel. In this case, two sound output devices are

attached to the computer, the built-in speaker and a speaker attached to the SurfBoard DSP card. The highlighted icon shows which device is the **current sound output device.** All sounds produced by the Sound Manager are sent to that device for playback, unless you specify some other device when creating a sound channel. (See the description of `SndNewChannel` in the chapter "Sound Manager" for details on specifying an output device explicitly.)

Figure 1-6   The Sound Out control panel
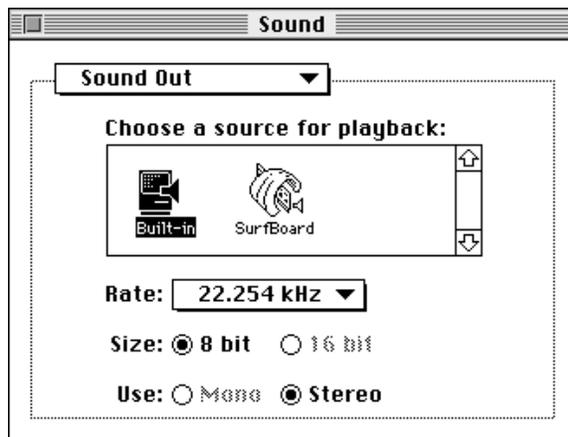
Note
This book shows the Sound control panels introduced with version 3.0 of the Sound Manager. Users can use the pop-up menu at the top of the panel to select one of four or more subpanels (Alert Sounds, Sound In, Sound Out, and Volumes). It's possible to add new subpanels to the Sound control panel. See the chapter on control panel extensions in the book *Inside Macintosh: Operating System Utilities.* ◆

You can play a sound by calling a Sound Manager routine such as `SysBeep` (to play the system alert sound), `SndPlay` (to play a sound stored in memory), or `SndStartFilePlay` (to play a sound stored in a file). The Sound Manager then issues one or more sound commands to the audio hardware. A **sound command** is an instruction to produce sound, modify sound, or otherwise assist in the overall process of sound production.

To ensure that sound commands are issued in the correct order, the Sound Manager uses a structure called a sound channel to store commands. A **sound channel** is associated with a first-in, first-out (FIFO) queue of sound commands. Queued commands are sent to the sound hardware through a sound output device component, a component that manages the last stage of communication with the audio hardware. Figure 1-7 shows how your application communicates, through the Sound Manager and the sound output device component, with the current sound output device.

Figure 1-7   The relation of the Sound Manager to the audio hardware

Commands

Sound Manager

Sound channel

Sound component
chain

Audio
hardware
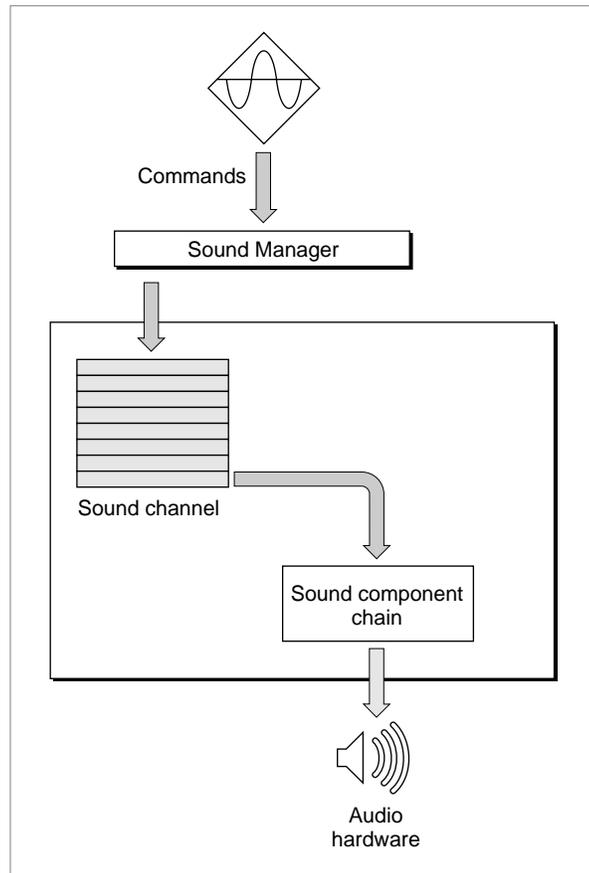
Note
This chapter does not discuss sound commands or channels in detail,
because you do not need to know about these details to play sound data
stored in sound resources or sound files. This chapter describes only how
to play and record sampled sounds. For more information on sound
channels and sound commands, see the chapter "Sound Manager" in this
book. ◆

You can play sounds either synchronously or asynchronously. When you play a sound
synchronously, the Sound Manager alone has control over the CPU while it executes
commands in a sound channel. Your application does not continue executing until the
sound has finished playing. When you play a sound **asynchronously,** your application
can continue other processing while the sound is playing. This chapter shows how to
play sounds only synchronously. To learn how to play sounds asynchronously, see the
chapter "Sound Manager" in this book.

Sometimes it is necessary to bypass the queue of sound commands. If, for example, you
want to stop all sound production on a particular channel immediately, it would be
counterproductive to put the command into the sound channel because that command

wouldn't be processed until any others already in the queue were processed. You can send sound commands directly to the hardware component, as shown in Figure 1-8.

When you bypass the sound channel in this way, any commands that are already queued but not yet sent to the sound output device component remain queued. You can, however, flush the channel at any time by sending the Sound Manager the appropriate request.

Figure 1-8　　Bypassing the command queue
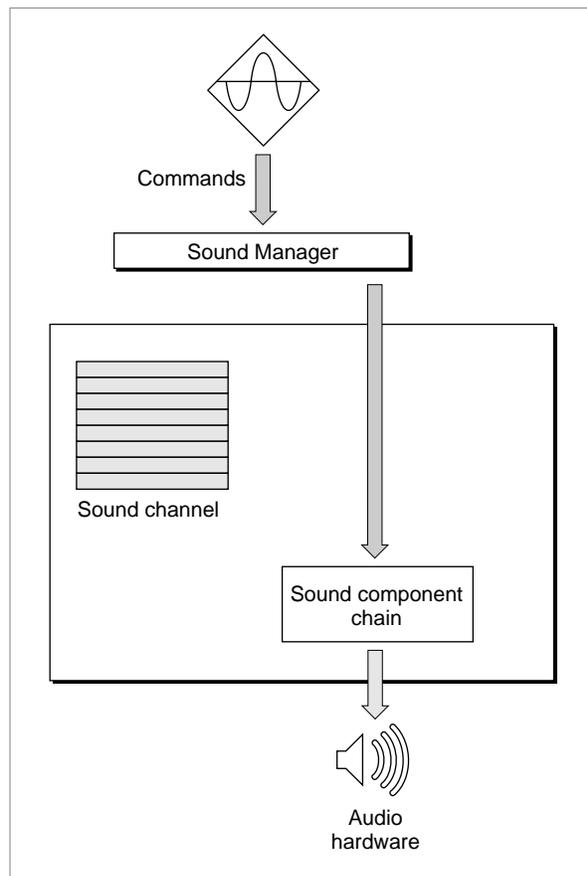
It's possible to have several channels of sound open at one time. The Sound Manager (using a sound-mixing component called the **Apple Mixer component**) mixes together the data coming from all open sound channels and sends a single stream of sound data to the current sound output device. This allows a single application to play two or more sounds at once. It also allows multiple applications to play sounds at the same time, as illustrated in Figure 1-9.

Figure 1-9   Mixing multiple channels of sampled sound
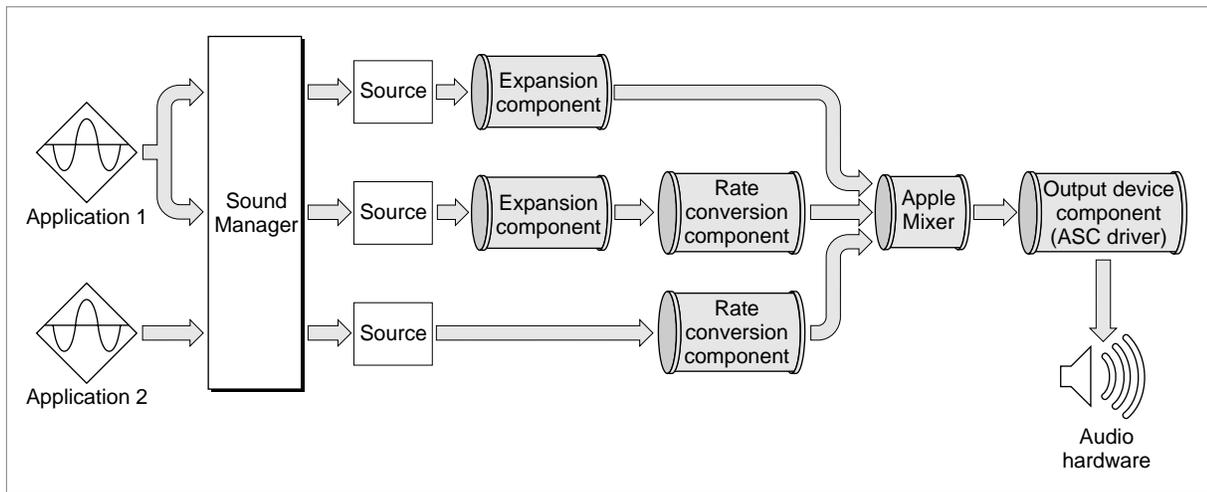
The Sound Manager was first released for all Macintosh computers as part of system software version 6.0. System software versions 6.0.7 and later include an **enhanced Sound Manager** (that is, version 2.0) that provides routines for continuous play from disk, sound mixing, and audio compression and expansion. System software versions 6.0.7 and later also include the Sound Input Manager, which allows for recording sounds through either a built-in microphone or some other sound input device.

More recent versions of the Sound Manager significantly improve the performance of the Sound Manager's operations and extends its capabilities. Version 3.0 of the Sound Manager is as much as two to three times more efficient than previous versions, which allows your application to do more processing while a sound is playing. In addition, version 3.0 of the Sound Manager provides three important new capabilities:

■ **Support for 16-bit audio samples.** Versions of the Sound Manager earlier than version 3.0 support only 8-bit monophonic or stereo audio samples with sample rates up to 22 kHz. The Sound Manager version 3.0 supports 16-bit stereo audio samples with sample rates up to 64 kHz, thereby allowing your application to produce CD-quality sound. Moreover, the Sound Manager version 3.0 automatically converts 16-bit samples into 8-bit samples on Macintosh computers that do not have the hardware to output 16-bit sounds.

■ **Support for non-Apple audio hardware.** The Sound Manager version 3.0 and later use a sound architecture that allows support for third-party audio hardware. This allows a user to install audio hardware capable of recording and producing CD-quality sound. Versions 3.0 and later also include a new Sound control panel that allows the user to redirect sound output to any available audio hardware.

■ **Support for plug-in codecs.** Versions of the Sound Manager earlier than version 3.0 support audio compression and expansion only at ratios of 3:1 and 6:1. The Sound Manager version 3.0 provides support for other compressed audio data formats by allowing plug-in audio compression and expansion components (or codecs).

You provide support for your own sound output devices or for your own compression and decompression algorithms by writing an appropriate sound component. See the chapter "Sound Components" later in this book for complete details.

The Sound Manager version 3.0 is supported only on Macintosh computers with an ASC or comparable hardware. In particular, the Sound Manager version 3.0 is not supported on Macintosh Classic, Macintosh Plus, or Macintosh SE computers. As a result, you should always test whether the specific capabilities you want to use are present before attempting to use them. You can use the Gestalt function to do this, as illustrated in "Checking For Sound-Recording Equipment" beginning on page 41 and in "Checking For Speech Capabilities" beginning on page 45.

This book describes the capabilities and programming interfaces of version 3.0 of the Sound Manager. Many of the techniques described here can also be used with earlier versions of the Sound Manager, but some cannot. Make sure to test your application thoroughly with all versions of the Sound Manager you want to run under.

Sound Recording

The Sound Input Manager provides the ability to record and digitally store sounds in a device-independent manner. You can create a resource or a file containing a recorded sound simply by calling either the SndRecord function or the SndRecordToFile function. You can then use the recorded sound in any way appropriate to your application.

The sound input and storage routines can be used with any available sound input hardware for which there is an appropriate device driver. A user can select from among the available sound input devices through the **Sound In control panel,** shown in Figure 1-10.

Figure 1-10   The Sound In control panel
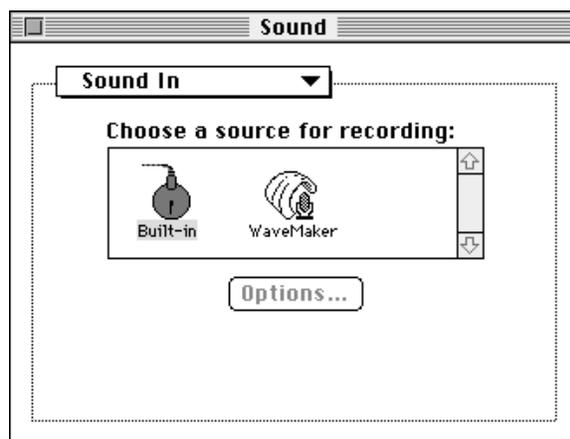
The available sound input devices are listed in the center of the panel. The control panel lists a device if its driver has previously registered itself with the Sound Input Manager and has provided a name and device icon. In Figure 1-10, two sound input devices are available, a device named Built-in and a device named WaveMaker. The highlighted icon shows which device is the **current sound input device.**

The **Alert Sounds control panel** lists the available system alert sounds, as illustrated in Figure 1-11.

Figure 1-11   The Alert Sounds control panel
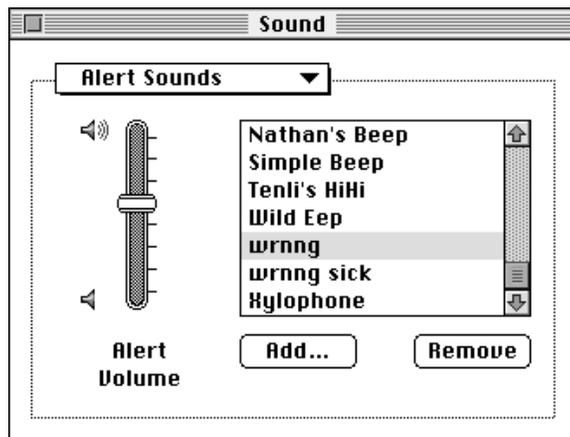
The Alert Sounds control panel also includes two buttons, Add and Remove. These buttons allow the user to add sounds to and remove sounds from the list of available system alert sounds. The Add button is used to record a new alert sound and add it to the list. Clicking the Add button causes the Sound Input Manager to display a sound recording dialog box (described later in this section). Clicking the Remove button causes the Sound Input Manager to remove the selected alert sound from the list. The user can achieve the same effect by selecting a sound and then choosing the Clear command in the Edit menu. If no sound input drivers are installed in the system, these two buttons do not appear.

If the user records a sound using the Alert Sounds control panel, the recorded sound is saved as a resource of type 'snd ' in the System file. That sound then appears in the list of available alert sounds. Note that the Alert Sounds control panel supports the standard Edit menu commands on sounds stored in the System file. The Cut command copies the selected sound to the Clipboard and removes it from the list of system alert sounds. The Copy command just copies the selected sound to the Clipboard. The Paste command takes a sound copied from the Clipboard and places it in the list of available alert sounds. If your application allows users to manipulate sound resources, it should support the copying and pasting of sound resources through the Clipboard. However, the Undo command does not work with sound-related editing operations.

The Sound Input Manager provides two high-level routines that allow your application to record sounds from the user and store them in memory or in a file. When you call either `SndRecord` or `SndRecordToFile`, the Sound Input Manager presents a **sound recording dialog box** to the user, illustrated in Figure 1-12.

Figure 1-12   The sound recording dialog box
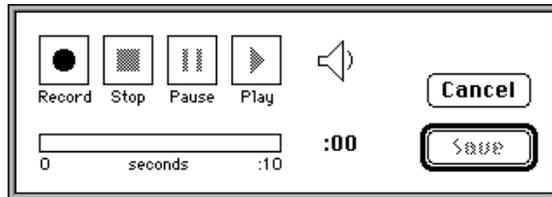
Using the controls in this dialog box, the user can start, pause, resume, and stop recording on the currently selected sound input device. The user can also play back the recorded sound. The time indicator bar provides an indication of the current length of the recorded sound.

When the user clicks the Save button after initiating a recording from the Sound control panel, another dialog box appears asking the user to give the sound a name. Unless the user cancels the save operation at that point, the Sound control panel saves the recorded sound into a sound resource in the System file. Note that if your application can save recorded sound resources, the `SndRecord` function does not present the dialog box that allows the user to name the sound and does not automatically save the recorded sound into a resource file. Your application must provide code to accomplish these tasks.

Sound Resources

Resources of type `'snd '` (also called **sound resources**) can contain both sound commands and sound data, and are widely used by sound-producing applications. 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. The user can select the current system alert sound with the Alert Sounds control panel, as illustrated in Figure 1-11. More generally, you can load a sound resource into memory and then play it by calling the `SndPlay` function.

Note
If you do not use the sound-recording routines provided by the Sound Input Manager, you must know the structure of `'snd '` resources before you can create them. For information on this, see the chapter "Sound Manager" in this book. You can also use the `SetupSndHeader` function, described in the chapter "Sound Input Manager" in this book, to help you create an `'snd '` resource. ◆

The Sound Manager can read sound resources in two formats, format 1 or format 2. However, the format 2 `'snd '` resource is obsolete, so your application should use format 1 `'snd '` resources. For more information on the differences between format 1 and format 2 `'snd '` resources, see the chapter "Sound Manager" in this book.

The format 1 `'snd '` resource is the most general kind of sound resource. A format 1 `'snd '` resource can contain a sequence of Sound Manager commands and associated sound data (such as wave-table data or a sampled sound header that both describes a digitally recorded sound and includes the sampled-sound data itself). Your application can produce sounds simply by passing a handle to that resource to the `SndPlay` function, which opens a sound channel and sends the commands and data contained in the resource into the channel. Alternatively, a format 1 `'snd '` resource might contain a sequence of commands that describe a sound, without providing any other sound data. For example, such a resource could contain a command that alters the amplitude (or loudness) of sound playing on a channel. In this case, your application can use the `SndPlay` function to execute the commands on any channel.

Sound Files

Although most sampled sounds that you want your application to produce can be stored as sound resources, there are times when it is preferable to store sounds in **sound files.** For example, it is usually easier for different applications to share files than it is to share resources. So, if you want your application to play a sampled sound created by another application (or if you want other applications to be able to play a sampled sound created by your application), it might be better to store the sampled-sound data in a file, not in a resource. Similarly, if you are developing versions of your application that run on other operating systems, you might need a method of storing sounds that is independent of the Macintosh Operating System and its reliance on resources to store data. Generally, it is easier to transfer data stored in data files from one operating system to another than it is to transfer data stored in resources.

There are other reasons you might want to store some sampled sounds in files and not in resources. If you have a very large sampled sound, it might not be possible to create a resource large enough to hold all the audio data. Resources are limited in size by the structure of resource files (and in particular because offsets to resource data are stored as 24-bit quantities). Sound files, however, can be much larger because the only size limitations are those imposed by the file system on all files. If the sampled-sound data for some sound occupies more than about a half megabyte of space, you should probably store the sound as a file.

To address these various needs, Apple and several third-party developers have defined two sampled-sound file formats, known as the **Audio Interchange File Format (AIFF)** and the **Audio Interchange File Format Extension for Compression (AIFF-C).** The names emphasize that the formats are designed primarily as data interchange formats. However, you should find both AIFF and AIFF-C flexible enough to use as data storage formats as well. Even if you choose to use a different storage format, your application should be able to convert to and from AIFF and AIFF-C if you want to facilitate sharing

of sound data among applications. AIFF format files have file type 'AIFF' and AIFF-C format files have file type 'AIFC'.

Note

Do not confuse AIFF and AIFF-C files (referred to in this chapter as *sound files*) with Finder sound files. Each **Finder sound file** contains a sound resource that plays when the user double clicks on the file in the Finder (or selects the file and chooses Open from the File menu). A user can create a Finder sound file by dragging a sound out of the System file, and a user can drag a Finder sound file into the System file to add the file's sound to the list of available system alert sounds. You can create a Finder sound file by creating a file of type 'sfil' with a creator of 'movr' and placing in the file a single sound resource. You can play such a file by using Resource Manager routines to open the Finder sound file and then by using the SndPlay function to play the single sound resource contained in it. ◆

The main difference between the AIFF and AIFF-C formats is that AIFF-C allows you to store either compressed or noncompressed audio data, whereas AIFF allows you to store noncompressed audio data only. The AIFF-C format is more general than the AIFF format and is easier to modify. The AIFF-C format can be extended to handle new compression types and application-specific data. As a result, if your application reads or writes sound files, it should be able to handle both AIFF and AIFF-C files. Table 1-1 summarizes the capabilities of the AIFF and AIFF-C file formats.

Table 1-1   AIFF and AIFF-C capabilities

File type	Read sampled	Read compressed	Write sampled	Write compressed
AIFF	Yes	No	Yes	No
AIFF-C	Yes	Yes	Yes	Yes

The enhanced Sound Manager includes **play-from-disk** routines that allow you to play AIFF and AIFF-C files continuously from disk even while other tasks execute. You might think of the play-from-disk routines as providing you with the ability to install a "tape player" in a sound channel. Once the sound begins to play, it continues uninterrupted until it finishes or until an application pauses or stops it.

You can play a sampled sound stored in a file of type AIFF or AIFF-C by opening the file and passing its file reference number to the SndStartFilePlay function. If the file is of type AIFF-C and the data is compressed, then the data is automatically expanded during playback. The SndStartFilePlay function works like the SndPlay function but does not require the entire sound to be in RAM at one time. Instead, the Sound Manager uses two buffers, each of which is smaller than the sound itself. The Sound Manager plays one buffer of sound while filling the other with data from disk. After it finishes playing the first buffer, the Sound Manager switches buffers, and plays data in the second while refilling the first. This **double buffering** technique minimizes RAM usage at the expense

of additional disk overhead. As a result, `SndStartFilePlay` is ideal for playing very large sounds.

The disk overhead incurred when using `SndStartFilePlay` is relatively light, and most mass-storage devices currently available for Macintosh computers have response times that are good enough that `SndStartFilePlay` can retrieve audio data from disk and play a sound without audible gaps. There are no limits on the number of concurrent disk-based sampled-sound playbacks other than those imposed by processor speed and disk capability. On machines with sufficient CPU resources, several continuous playbacks can occur at once. Disk fragmentation can affect the performance of playing sampled-sound files from disk. In addition, playing multiple sounds from the same hard disk may degrade overall performance.

The Sound Manager currently supports continuous play from disk only on certain Macintosh computers. You should use the `Gestalt` function to determine whether a specific machine supports play from disk. Also, if a sound channel is being used for continuous play from disk, then no other sound commands can be sent to that channel.

Speech Generation

The Speech Manager converts text into sound data, which it passes to the Sound Manager to play through the current sound output device. The Speech Manager's interaction with the Sound Manager is transparent to your application, so you don't need to be familiar with the Sound Manager to take advantage of the Speech Manager's capabilities. This section provides an overview of the Speech Manager and outlines the process that the Speech Manager uses to convert text into speech.

Figure 1-13 illustrates the speech generation process. Your application can initiate speech generation by passing a string or a buffer of text to the Speech Manager. The Speech Manager is responsible for sending the text to a **speech synthesizer,** a component that contains executable code that manages all communication between the Speech Manager and the Sound Manager. A synthesizer is usually contained in a resource in a file within the System Folder. The synthesizer uses built-in dictionaries and pronunciation rules to help determine how to pronounce text. Your application can use the default system voice to generate speech; it can also specify that some other available voice be used for speech generation.

As Figure 1-13 suggests, the Speech Manager is a dispatching mechanism that allows your application to take advantage of the capabilities of whatever speech synthesizers, voices, and hardware are installed. The Speech Manager itself does not do any of the work of converting text into speech; it just provides a convenient programming interface that manages access to speech synthesizers and, indirectly, to the sound hardware. The Speech Manager uses the Component Manager to access whatever speech synthesizers are available and allows applications to take maximum advantage of a computer's speech facilities without knowing what those facilities are.

Figure 1-13   The speech generation process
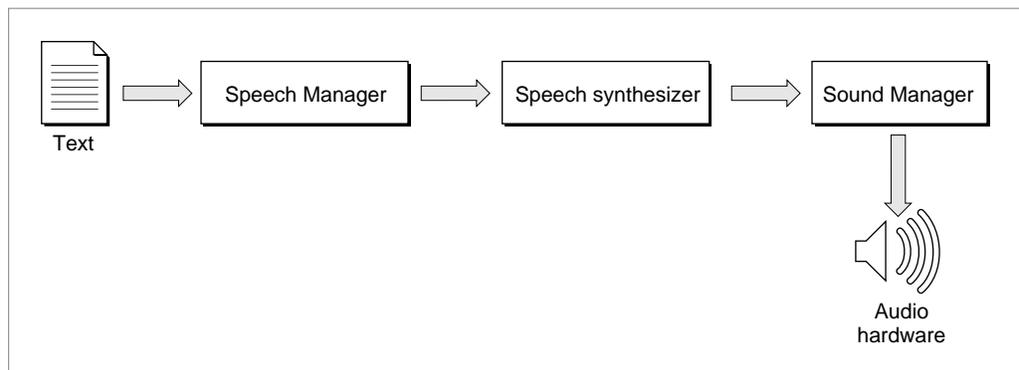
Note

The Component Manager is described in *Inside Macintosh:*
More Macintosh Toolbox, but you do not need to be familiar with
it to use the Speech Manager. ◆

A speech synthesizer can include one or more voices, as illustrated in Figure 1-14. Just as
different people's voices have different tonal qualities, so too can different voices in a
synthesizer. A synthesized voice might sound male or female, and might sound like an
adult or child. Some voices sound distinctively synthetic, while others sound more like
real people. As speech synthesizing technology develops, the voices that your application
can access are likely to sound more and more human. Because the Speech Manager's
routines work on all voices and synthesizers, you will not need to rewrite your
application to take advantage of improvements in speech technology.

Figure 1-14   The Speech Manager and multiple voices
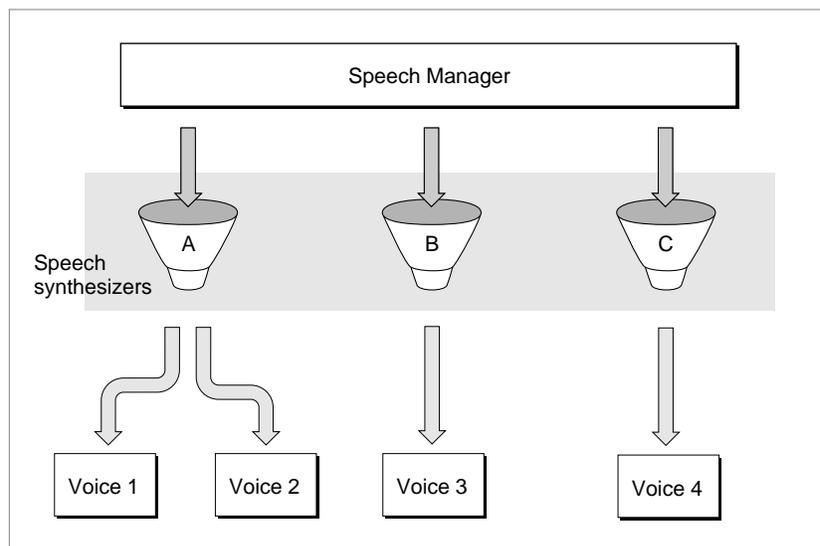
Any given person has only one voice, but can alter the characteristics of his or her speech in a number of different ways. For example, a person can speak slowly or quickly, and with a low or a high pitch. Similarly, the Speech Manager provides routines that allow you to modify these and other speech attributes, regardless of which voice is in use.

To indicate to the Speech Manager which voice or attributes you would like it to use in generating speech, your application must use a speech channel. A **speech channel** is a data structure that the Speech Manager uses when processing text; it can be associated with a particular voice and particular speech attributes. Because multiple speech channels can coexist, your application can create several different vocal environments (to simulate a conversation, for example). Because a synthesizer can be associated with only one language and region, your application would need to create a separate speech channel to process each language in bilingual or multilingual text. (Currently, however, only English-producing synthesizers are available.)

Different speech channels can even generate speech simultaneously, subject to processor capabilities and Sound Manager limitations. This capability should be used with restraint, however, because it can be hard for the user to understand any speech when more than one channel is generating speech at a time. Also, your application should in general generate speech only at the specific request of the user and should allow the user to turn off speech output. At the very least, your application should include an option that allows the user to view text instead of hearing it. Some users might have trouble understanding speech generated by the Speech Manager, and others might be hearing-impaired. Even users who are able to clearly understand computer-synthesized speech might prefer to read rather than hear.

In general, your application does not need to know which speech synthesizer it is using. You can obtain a list of all available voices, but in most cases, you do not need to be concerned with which speech synthesizer a voice is associated. Sometimes, however, a speech synthesizer may provide special capabilities beyond that provided by the Speech Manager. For example, a speech synthesizer might allow you to select an option to read numbers in a nonstandard way. The Speech Manager allows you to determine which synthesizer is associated with a voice for these circumstances, and provides hooks that allow your application to take advantage of synthesizer-specific capabilities.

In general, however, your application can achieve the best results by making no assumptions about which synthesizers might be available. The user of a 2 MB Macintosh Classic® might use a synthesizer with low RAM requirements, while the user of a 20 MB Macintosh Quadra 950 might take advantage of a synthesizer that provides better audio quality at the expense of memory usage. The Speech Manager makes it easy to accommodate both kinds of users.

The most basic use of the Speech Manager is to convert a text string into speech. The `SpeakString` function, described in "Generating Speech From a String" beginning on page 46, lets you do this even without allocating a speech channel. The chapter "Speech Manager" in this book describes how you can customize the quality of speech output to make it easier to understand and how you can obtain more control over speech by allocating speech channels and embedding commands within text strings.

The User Interface for Sound

As you have seen, the Macintosh system software provides you with a wide array of easy-to-use sound-input and sound-output services. With very little programming, you can

- play the user's system alert sound or any sound contained in a sound resource or file

- record sounds through the available sound-input hardware

- convert text into speech

The system software has already defined a set of user interface elements and metaphors that are designed to facilitate the integration of sound into the Macintosh graphical user interface. In general, you should use the existing system software services to present the standard interface elements designed by Apple. For example, if you want to have the user record through the available sound-input hardware, you can call the `SndRecord` function, which displays the sound recording dialog box (shown in Figure 1-12 on page 31). That dialog box contains controls that are modelled on the buttons typically found on an audio tape recorder or a video cassette recorder. In this way, the system software draws on the user's knowledge of how to operate a tape recorder and uses it as a metaphor for recording sounds on Macintosh computers.

The system software also provides visual representations of sounds themselves. In some cases, sounds are represented by their names only, as in the Alert Sounds control panel (shown in Figure 1-11 on page 30). In other cases, sounds are represented by icons. For example, the icon for a Finder sound looks like the one shown in Figure 1-15. All Finder sounds are represented by the same icon; they are distinguished from each other by their names.

Figure 1-15   An icon for a Finder sound

Xylophone

If the user copies or cuts a sound from the available system alert sounds and then pastes the sound into the Scrapbook, the sound is shown as in Figure 1-16.

Figure 1-16   A sound in the Scrapbook
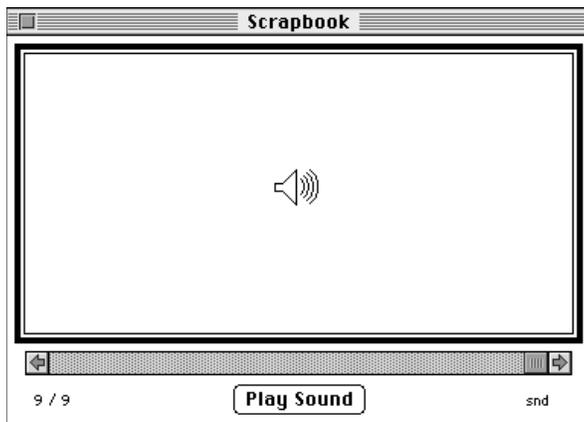
As you can see, the metaphor in both cases is that of a speaker, a sound-producing device familiar to most computer users. If you need to design icons to represent sounds created by your application, you might want to use (or suitably adapt) these existing metaphors. For example, if your application supports document annotations with recorded voices or other sounds, you can display a speaker icon within the document. Clicking or double-clicking the icon should result in playing the sound.

Keep in mind that applications that play sound should allow users to turn off sound output, because there might be users who object to it or environments where it is inappropriate. Also, there might be cultural biases or preferences associated with certain sounds. Thus, if your application plays specific sounds, you should store them as resources, which can be easily modified for local regions, or if they are very large, in sound files, which you can replace easily during localization.

Using Sound on Macintosh Computers

This section describes the most basic ways of using the Sound Manager, the Sound Input Manager, and the Speech Manager. In particular, it provides source code examples that show how to produce an alert sound, play a sound resource, play a sound file, determine whether your application can access sound recording equipment, record a sound resource, record a sound file, and convert a text string to spoken words.

Producing an Alert Sound

You can produce a **system alert sound** to catch the user's attention by calling the `SysBeep` procedure. The `SysBeep` procedure is a Sound Manager routine that plays the alert sound selected by the user in the Alert Sounds control panel. Here's an example of calling `SysBeep`:

```
IF myErr <> noErr THEN
   SysBeep(30);
```

You must supply a parameter when you call the SysBeep procedure, even though the Sound Manager ignores that parameter in most cases. All system alert sounds are stored as format 1 'snd ' resources in the System file and are played by the Sound Manager. There is one instance in which the number passed to SysBeep is not ignored: if the user has selected the Simple Beep as the system alert sound on some Macintosh computers (for example, a Macintosh Plus or Macintosh SE), the beep is generated by code stored in ROM rather than by the Sound Manager, and the duration parameter is interpreted in ticks (sixtieths of a second).

The SysBeep procedure has no effect if an application has disabled the system alert sound. You might do this to prevent the system alert sound from interrupting some other sound. For information on enabling and disabling the system alert sound, see the chapter "Sound Manager" in this book.

You should not call the SysBeep procedure at interrupt time, because doing so causes the Sound Manager to attempt to allocate memory and load a resource.

Note

If your primary use of the SysBeep procedure is to alert the user of important or abnormal occurrences, it might be preferable to use the Notification Manager. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for complete details on alerting the user. ◆

Playing a Sound Resource

You can play a sound stored in a resource by calling the SndPlay function, which requires a handle to an existing 'snd ' resource. An 'snd ' resource contains sound commands that play the desired sound. The 'snd ' resource might also contain sound data. If it does (as in the case of a sampled sound), that data might be either compressed or noncompressed. SndPlay decompresses the data, if necessary, to play the sound. Listing 1-1 illustrates how to play a sound resource.

Listing 1-1   Playing a sound resource with SndPlay

```
FUNCTION MyPlaySndResource (mySndID: Integer): OSErr;
CONST
   kAsync = TRUE;                     {for asynchronous play}
VAR
   mySndHandle:   Handle;             {handle to an 'snd ' resource}
   myErr:         OSErr;
BEGIN
   mySndHandle := GetResource('snd ', mySndID);
   myErr := ResError;                 {remember any error}
   IF mySndHandle <> NIL THEN         {check for a NIL handle}
```

```
    BEGIN
        HLock(mySndHandle);              {lock the sound data}
        myErr := SndPlay(NIL, mySndHandle, NOT kAsync);
        HUnlock(mySndHandle);            {unlock the sound data}
        ReleaseResource(mySndHandle);
    END;
    MyPlaySndResource := myErr;       {return the result}
END;
```

When you pass `SndPlay` a `NIL` sound channel pointer in its first parameter, the Sound Manager automatically allocates a sound channel (in the application's heap) and then disposes of it when the sound has completed playing. Note, however, that when your application does pass `NIL` as the pointer to a sound channel, the third parameter to `SndPlay` is ignored; the sound plays synchronously even if you specify that you want it to play asynchronously.

IMPORTANT

The handle you pass to `SndPlay` must be locked for as long as the sound is playing. ▲

Playing a Sound File

You can initiate and control a playback of sampled sounds stored in a file using the `SndStartFilePlay`, `SndPauseFilePlay`, and `SndStopFilePlay` functions. You use `SndStartFilePlay` to initiate the playing of a sound file. If you allocate your own sound channel and specify that play be asynchronous, you can then use the `SndPauseFilePlay` and `SndStopFilePlay` functions to pause, resume, and stop sound files that are playing. The chapter "Sound Manager" in this book describes these two functions in detail.

To play a sampled sound that is contained in a file, you pass `SndStartFilePlay` the file reference number of the file to play. The sample should be stored in either AIFF or AIFF-C format. If the sample is compressed, it is automatically expanded during playback. If you specify `NIL` as the sound channel, then `SndStartFilePlay` allocates memory for a channel internally. Listing 1-2 defines a function that plays a file specified by its file reference number.

Listing 1-2   Playing a sound file with `SndStartFilePlay`

```
FUNCTION MyPlaySoundFile (myFileRefNum: Integer): OSErr;
CONST
    kAsync = TRUE;                  {for asynchronous play}
    kBufferSize = 20480;        {20K play buffer}
VAR
    myErr:    OSErr;
BEGIN
```

```
        myErr := SndStartFilePlay(NIL, myFileRefNum, 0, kBufferSize,
                                   NIL, NIL, NIL, NOT kAsync);
        MyPlaySoundFile := myErr;
    END;
```

To allow the Sound Manager to handle all memory allocation automatically, you should pass NIL as the first and fifth parameters to SndStartFilePlay, as done in Listing 1-2. The first NIL specifies that you want SndStartFilePlay to allocate a sound channel itself. The NIL passed as the fifth parameter specifies that SndStartFilePlay should automatically allocate buffers to play the sound. The SndStartFilePlay function then allocates two buffers, each half the size specified in the fourth parameter; if the fourth parameter is 0, the Sound Manager chooses a default size for the buffers.

The third parameter passed to SndStartFilePlay here is set to 0. That parameter is used only when playing sound resources from disk.

The seventh parameter to SndStartFilePlay allows you to specify a routine to be executed when the sound finishes playing. This is useful only for asynchronous play. In Listing 1-2, the value NOT kAsync (that is, FALSE) is passed as the eighth parameter to SndStartFilePlay to request synchronous playback. SndStartFilePlay would return a badChannel result code if you request asynchronous playback because MyPlaySoundFile does not allocate a sound channel.

Checking For Sound-Recording Equipment

Before allowing a user to record a sound, you must ensure that sound-recording hardware and software are installed. You can record sound through the microphone built into several Macintosh models, or through third-party sound input devices. Because low-level sound input device drivers handle communication between your application and the sound recording hardware, you do not need to know what type of microphone is available. Listing 1-3 defines a function that determines whether sound recording hardware is available.

Listing 1-3   Determining whether sound recording equipment is available

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

The `MyHasSoundInput` function defined in Listing 1-3 uses the `Gestalt` function to determine whether sound input hardware is available and usable on the current Macintosh computer. `MyHasSoundInput` tests the `gestaltHasSoundInputDevice` bit and returns `TRUE` if you can record sounds. `MyHasSoundInput` returns `FALSE` if you cannot record sounds (either because no sound input device exists or because the Sound Input Manager is not available).

Note

For more information on the `Gestalt` function, see *Inside Macintosh: Operating System Utilities.* ◆

Recording a Sound Resource

You can record sounds from the current input device by using the `SndRecord` function. The `SndRecord` function presents the sound recording dialog box. When calling `SndRecord`, you need to provide a handle to a block of memory where the incoming data should be stored. If you pass the address of a `NIL` handle, however, the Sound Input Manager allocates a large block of space in your application heap and resizes it when the recording stops. Listing 1-4 illustrates how to call `SndRecord`.

Listing 1-4   Recording through the sound recording dialog box

```
PROCEDURE MyRecordThruDialog (VAR mySndHandle: Handle);
VAR
   myErr:        OSErr;
   myCorner:     Point;
BEGIN
   MyGetTopLeftCorner(myCorner);
   mySndHandle := NIL;       {use default memory allocation}
   myErr := SndRecord(NIL, myCorner, siBestQuality, mySndHandle);
   IF (myErr <> noErr) AND (myErr <> userCanceledErr) THEN
      DoError(myErr);
END;
```

If the user cancels sound recording, then the `SndRecord` function returns the result code `userCanceledErr`. The `MyRecordThruDialog` procedure defined in Listing 1-4 returns a `NIL` sound handle if the user cancels recording.

If you pass a sound handle that is not `NIL` as the fourth parameter to the `SndRecord` function, the Sound Input Manager derives the maximum time of recording from the amount of space reserved by that handle. The handle is resized on completion of the recording.

The first parameter in the call to `SndRecord` is the address of a filter procedure that determines how user actions in the dialog box are filtered. In Listing 1-4, no filter procedure is desired, so the parameter is specified as `NIL`. For information

on filter procedures, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

The second parameter in the call to `SndRecord` is the desired location (in global coordinates) of the upper-left corner of the dialog box. For example, the Sound control panel displays the dialog box near the control panel. Your application might place the dialog box elsewhere (for example in the standard alert position on the main screen). For more information on centering dialog boxes, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

The third parameter in the call to `SndRecord` specifies the quality of the recording. Currently three values are supported:

```
CONST
    siBestQuality     = 'best';   {the best quality available}
    siBetterQuality   = 'betr';   {a quality better than good}
    siGoodQuality     = 'good';   {a good quality}
```

The precise meanings of these constants are defined by the current sound-input device driver. The constant `siBestQuality` indicates that you want the highest quality recorded sound, usually at the expense of increased storage space (possibly because no compression is performed on the sound data). The constant `siGoodQuality` indicates that you are willing to sacrifice audio quality if necessary to minimize the amount of storage space required (typically this means that 6:1 compression is performed on the sound data). For most voice recording, you should specify `siGoodQuality`. The constant `siBetterQuality` defines a quality and storage space combination that is between those provided by the other two constants.

You could play the sound recorded using the `MyRecordThruDialog` procedure defined in Listing 1-4 by calling `SndPlay` and passing it the sound handle `mySndHandle`. That handle refers to some data in memory that has the structure of an `'snd '` resource, but it is not a handle to an existing resource. To save the recorded data as a resource, you can use the Resource Manager. Listing 1-5 calls the `MyRecordThruDialog` procedure and then uses the Resource Manager to save the recorded data as a resource in an open resource file.

Listing 1-5   Recording a sound resource

```
PROCEDURE MyRecordSndResource (resFileRefNum: Integer);
CONST
   kMinSysSndRes = 0;                  {lowest reserved 'snd ' resource ID}
   kMaxSysSndRes = 8191;               {highest reserved ID}
VAR
   myPrevResFile:        Integer;      {current resource file}
   mySndHandle:          Handle;       {handle to resource data}
   myResID:              LongInt;      {ID of resource}
   myResName:            Str255;       {name of resource}
```

```
   myErr:                   OSErr;
BEGIN
   myPrevResFile := CurResFile;      {remember current resource file}
   UseResFile(resFileRefNum);        {temporarily switch resource files}
   MyRecordThruDialog(mySndHandle);  {record via standard interface}
   IF mySndHandle <> NIL THEN
   BEGIN                             {recording finished successfully}
      REPEAT                         {find acceptable resource ID number}
         myResID := Unique1ID('snd ');
      UNTIL (myResID < kMinSysSndRes) OR (myResID > kMaxSysSndRes);

      MyGetSoundName(myResName);     {get name for sound resource}
                                     {add resource to file}
      AddResource(mySndHandle, 'snd ', myResID, myResName);
      myErr := ResError;
      IF myErr = noErr THEN
      BEGIN
         UpdateResFile(resFileRefNum);    {update resource file}
         myErr := ResError;
      END;
      IF myErr <> noErr THEN
         DoError(myErr);
   END;
   UseResFile(myPrevResFile);        {restore previous resource file}
END;
```

The `MyRecordSndResource` procedure defined in Listing 1-5 takes as a parameter the reference number of an open resource file to which you wish to record. The procedure makes that resource file the current resource file and, after recording, reverts to what was previously the active resource file. Note that you should not record to your application's resource fork, because applications that write to their own resource forks cannot be used by multiple users at once over a network. For more information on reference numbers for resource files, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

The `MyRecordSndResource` procedure first presents the sound recording dialog box by calling the `MyRecordThruDialog` procedure defined in Listing 1-4 on page 42. If that procedure returns a valid sound handle, `MyRecordSndResource` finds an acceptable resource ID for the resource file and then calls a procedure that returns a name for the resource (perhaps by presenting a dialog box that asks the user to name the sound). Finally, `MyRecordSndResource` adds the resource to the specified resource file and updates that file by calling the Resource Manager procedure `UpdateResFile`.

Recording a Sound File

To record a sound directly into a file, you can call the `SndRecordToFile` function, which works exactly like `SndRecord` except that you pass it the file reference number of an open file instead of a handle to some memory. When `SndRecordToFile` exits successfully, that file contains the recorded audio data in AIFF or AIFF-C format. You can then play the recorded sound by passing that file reference number to the `SndStartFilePlay` function. (See Listing 1-2 on page 40 for a sample function that uses the `SndStartFilePlay` function.) Listing 1-6 defines a procedure that records a sound into a file using `SndRecordToFile`.

Listing 1-6   Recording a sound file

```
PROCEDURE MyRecordSoundFile (myFileRefNum: Integer);
VAR
   myErr:      OSErr;
   myCorner:   Point;
BEGIN
   MyGetTopLeftCorner(myCorner);
   myErr := SndRecordToFile(NIL, myCorner, siBestQuality, myFileRefNum);
   IF (myErr <> noErr) AND (myErr <> userCanceledErr) THEN
      DoError(myErr);
END;
```

The `SndRecordToFile` function records the sound in the file specified in its fourth parameter. You must open the file before calling the `MyRecordSoundFile` procedure, and you must close the file after calling it. For more information on creating, opening, and closing files, see the chapter "Introduction to File Management" in *Inside Macintosh: Files.*

Checking For Speech Capabilities

Because the Speech Manager is not available in all system software versions, your application should always check for speech capabilities before attempting to use them. Listing 1-7 defines a function that determines whether the Speech Manager is available.

Listing 1-7   Checking for speech generation capabilities

```
FUNCTION MyHasSpeech: Boolean;
VAR
   myFeature:      LongInt;         {feature being tested}
   myErr:          OSErr;
BEGIN
   myErr := Gestalt(gestaltSpeechAttr, myFeature);
```

```
    IF myErr = noErr THEN           {test Speech Manager-present bit}
        MyHasSpeech := BTst(myFeature, gestaltSpeechMgrPresent)
    ELSE
        MyHasSpeech := FALSE;        {no speech features available}
END;
```

The `MyHasSpeech` function defined in Listing 1-7 uses the `Gestalt` function to determine whether the Speech Manager is available. The `MyHasSpeech` function tests the `gestaltSpeechMgrPresent` bit and returns `TRUE` if and only if the Speech Manager is present. If the `Gestalt` function cannot obtain the desired information and returns a result code other than `noErr`, the `MyHasSpeech` function assumes that the Speech Manager is not available and therefore returns `FALSE`.

Generating Speech From a String

It is easy to have the Speech Manager generate speech from a string stored as a variable of type `Str255`. The `SpeakString` function takes one parameter, the string to be spoken. `SpeakString` automatically allocates a speech channel, uses that channel to produce speech, and then disposes of the speech channel when speaking is complete. Speech generation is asynchronous, but because `SpeakString` copies the string you pass it into an internal buffer, you are free to release the memory you allocated for the string as soon as `SpeakString` returns.

Listing 1-8 show how you can use the `SpeakString` function to convert a string stored in a resource of type `'STR#'` into speech.

Listing 1-8   Using `SpeakString` to generate speech from a string

```
PROCEDURE MySpeakStringResource (myStrListID: Integer; myIndex: Integer);
VAR
    myString:       Str255;                                 {the string to speak}
    myErr:          OSErr;
BEGIN
    GetIndString(myString, myStrListID, myIndex);      {load the string}
    myErr := SpeakString(myString);                    {start speaking}
    IF myErr <> noErr THEN
        DoError(myErr);
END;
```

The `MySpeakStringResource` procedure defined in Listing 1-8 takes as parameters the resource ID of the `'STR#'` resource containing the string and the index of the string within that resource. `MySpeakStringResource` passes these values to the `GetIndString` procedure, which loads the string from the resource file into memory. `MySpeakStringResource` then calls the `SpeakString` function to convert the string into speech; if an error occurs, it calls an application-defined error-handling procedure.

The speech that the `SpeakString` function generates is asynchronous; that is, control returns to your application before the function finishes speaking the string. If you would like to generate speech synchronously, you can use `SpeakString` in conjunction with the `SpeechBusy` function, which returns the number of active speech channels, including the speech channel created by the `SpeakString` function.

Listing 1-9 illustrates how you can use `SpeechBusy` and `SpeakString` to generate speech synchronously.

Listing 1-9   Generating speech synchronously

```
PROCEDURE MySpeakStringResourceSync (myStrListID: Integer; myIndex: Integer);
VAR
   activeChannels:   Integer;              {number of active speech channels}
BEGIN
   activeChannels := SpeechBusy;        {find number of active channels}
   MySpeakStringResource(myStrListID, myIndex);        {speak the string}

   {Wait until channel is no longer processing speech.}
   REPEAT
   UNTIL SpeechBusy = activeChannels;
END;
```

The `MySpeakStringResourceSync` procedure defined in Listing 1-9 uses the `MySpeakStringResource` procedure defined in Listing 1-8 to speak a string. However, before calling `MySpeakStringResource`, `MySpeakStringResourceSync` calls the `SpeechBusy` function to determine how many speech channels are active. After the speech has begun, the `MySpeakStringResourceSync` function does not return until the number of speech channels active again falls to this level.

Note

Ordinarily, you should play speech asynchronously, to allow the user to perform other activities while speech is being generated. You might play speech synchronously if other activities performed by your application should not occur while speech is being generated. ◆

You can use the `SpeakString` function to stop speech being generated by a prior call to `SpeakString`. You might do this, for example, if the user switches to another application or closes a document associated with speech being generated. To stop speech, simply pass a zero-length string to the `SpeakString` function (or if you are programming in C, pass `NULL`).

Listing 1-10 shows how your application can stop speech generated by a call to the `SpeakString` function.

Listing 1-10   Stopping speech generated by `SpeakString`

```
PROCEDURE MyStopSpeech;
VAR
   myString:       Str255;              {an empty string}
   myErr:          OSErr;
BEGIN
   myString[0] := Char(0);              {set length of string to 0}
   myErr := SpeakString(myString);  {stop previous speech}
   IF myErr <> noErr THEN
      DoError(myErr);
END;
```

The `MyStopSpeech` procedure defined in Listing 1-10 sets the length byte of a string to 0 before calling the `SpeakString` function. To execute this code in some development systems, you need to ensure that range checking is disabled. Consult your development system's documentation for details on enabling and disabling range checking.

Sound Reference

This section describes the routines used in this chapter to illustrate basic sound producing and recording operations. These are high-level routines that you can use to play and record sound resources and sound files, and to convert text to speech. The routines described in this section also appear in the appropriate reference sections of the other chapters in this book.

For a description of sound-related data structures and other sound-related routines, see the chapters "Sound Manager," "Sound Input Manager," and "Speech Manager" in this book. For a detailed description of the formats of sound resources and sound files, see the chapter "Sound Manager" in this book.

Routines

This section describes the high-level system software routines that you can use to play and record sound resources and sound files, or to convert a text string to spoken words. These routines belong to the Sound Manager.

Playing Sounds

You can use the `SysBeep` procedure to play the system alert sound, the `SndPlay` function to play the sound stored in any `'snd '` resource, and the `SndStartFilePlay` function to play a sound file.

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 or for information on
reading and adjusting the system alert sound volume, see the chapter "Sound Manager"
in this book.

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 'snd ' resource. If the resource has not yet been
loaded, the SndPlay function fails and returns the resProblem result code. The handle
you pass in the sndHdl parameter must be locked for as long as the sound is playing
asynchronously.

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. 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.

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
"Playing a Sound Resource" on page 39. For more information on the SndPlay function,
see the chapter "Sound Manager" in this book.

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 (described in the chapter "Sound
Manager") 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.

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
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
siInvalidCompression	–223	Invalid compression type

SEE ALSO

For an example of how to play a sound file using the SndStartFilePlay function, see "Playing a Sound File" on page 40. For information on completion routines, see the chapter "Sound Manager" in this book.

Recording Sounds

The Sound Input Manager provides two high-level sound input routines, SndRecord and SndRecordToFile, for recording sound. These input routines are analogous to the two Sound Manager functions SndPlay and SndStartFilePlay. By using these high-level routines, you can be assured that your application presents a user interface that is consistent with that displayed by other applications recording sounds. Both SndRecord and SndRecordToFile attempt to record sound data from the sound input hardware currently selected in the Sound In control panel.

SndRecord

You can use the `SndRecord` function to record sound resources into memory.

```
FUNCTION SndRecord (filterProc: ProcPtr; corner: Point;
                    quality: OSType; VAR sndHandle: Handle):
                    OSErr;
```

filterProc
:   A pointer to an event filter function that determines how user actions in the sound recording dialog box are filtered (similar to the `filterProc` parameter specified in a call to the `ModalDialog` procedure). By specifying your own filter function, you can override or add to the default actions of the items in the dialog box. If `filterProc` isn't `NIL`, `SndRecord` filters events by calling the function that `filterProc` points to.

corner
:   The horizontal and vertical coordinates of the upper-left corner of the sound recording dialog box (in global coordinates).

quality
:   The desired quality of the recorded sound.

sndHandle
:   On entry, a handle to some storage space or `NIL`. On exit, a handle to a valid sound resource (or unchanged, if the call did not execute successfully).

DESCRIPTION

The `SndRecord` function records sound into memory. The recorded data has the structure of a format 1 `'snd '` resource and can later be played using the `SndPlay` function or can be stored as a resource. `SndRecord` displays a sound recording dialog box and is always called synchronously. Controls in the dialog box allow the user to start, stop, pause, and resume sound recording, as well as to play back the recorded sound. The dialog box also lists the remaining recording time and the current microphone sound level.

The `quality` parameter defines the desired quality of the recorded sound. Currently, three values are recognized for the `quality` parameter:

```
CONST
   siBestQuality        = 'best';   {the best quality available}
   siBetterQuality      = 'betr';   {a quality better than good}
   siGoodQuality        = 'good';   {a good quality}
```

The precise meanings of these parameters are defined by the sound input device driver. For Apple-supplied drivers, this parameter determines whether the recorded sound is to be compressed, and if so, whether at a 6:1 or a 3:1 ratio. The quality `siBestQuality` does not compress the sound and provides the best quality output, but at the expense of increased memory use. The quality `siBetterQuality` is suitable for most nonvoice recording, and `siGoodQuality` is suitable for voice recording.

The `sndHandle` parameter is a handle to some storage space. If the handle is NIL, the Sound Input Manager allocates a handle of the largest amount of space that it can find in your application's heap and returns this handle in the `sndHandle` parameter. The Sound Input Manager resizes the handle when the user clicks the Save button in the sound recording dialog box. If the `sndHandle` parameter passed to `SndRecord` is not NIL, the Sound Input Manager simply stores the recorded data in the location specified by that handle.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndRecord` function are

Trap macro	Selector
_SoundDispatch	$08040014

RESULT CODES

noErr	0	No error
userCanceledErr	–128	User canceled the operation
siBadSoundInDevice	–221	Invalid sound input device
siUnknownQuality	–232	Unknown quality

SEE ALSO

For an example of how to record a sound resource using the `SndRecord` function, see "Recording a Sound Resource" on page 42. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of event filter functions.

SndRecordToFile

You can use `SndRecordToFile` to record sound data into a file.

```
FUNCTION SndRecordToFile (filterProc: ProcPtr; corner: Point;
                          quality: OSType;
                          fRefNum: Integer): OSErr;
```

filterProc
: A pointer to a function that determines how user actions in the sound recording dialog box are filtered.
corner
: The horizontal and vertical coordinates of the upper-left corner of the sound recording dialog box (in global coordinates).

quality   The desired quality of the recorded sound. The values you can use for this
            parameter are described on page 53.

fRefNum   The file reference number of an open file to save the audio data in.

DESCRIPTION

The `SndRecordToFile` function works just like `SndRecord` except that it stores the
sound input data into a file. The resulting file is in either AIFF or AIFF-C format and
contains the information necessary to play the file by using the Sound Manager's
`SndStartFilePlay` function. The `SndRecordToFile` function is always called
synchronously.

Your application must open the file specified in the `fRefNum` parameter before calling the
`SndRecordToFile` function. Your application must close the file sometime after calling
`SndRecordToFile`.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndRecordToFile` function are

Trap macro	Selector
_SoundDispatch	$07080014

RESULT CODES

noErr	0	No error
userCanceledErr	–128	User canceled the operation
siBadSoundInDevice	–221	Invalid sound input device
siUnknownQuality	–232	Unknown quality

SEE ALSO

For an example of how to record a sound file using the `SndRecordToFile` function, see
"Recording a Sound File" on page 45. See the chapter "Dialog Manager" in *Inside
Macintosh: Macintosh Toolbox Essentials* for a complete description of event filter functions.

Generating and Stopping Speech

Your application can use the `SpeakString` function to generate speech or stop speech
currently being generated by `SpeakString`. By calling the `SpeechBusy` function before
and after a call to `SpeakString`, your application can determine when speaking is
complete. These routines belong to the Speech Manager.

SpeakString

You can use the `SpeakString` function to have the Speech Manager read a text string.

```
FUNCTION SpeakString (s: Str255): OSErr;
```

s   The string to be spoken.

DESCRIPTION

The `SpeakString` function attempts to speak the Pascal-style text string contained in the string `s`. Speech is produced asynchronously using the default system voice. When an application calls this function, the Speech Manager makes a copy of the passed string and creates any structures required to speak it. As soon as speaking has begun, control is returned to the application. The synthesized speech is generated asynchronously to the application so that normal processing can continue while the text is being spoken. No further interaction with the Speech Manager is required at this point, and the application is free to release the memory that the original string occupied.

If `SpeakString` is called while a prior string is still being spoken, the sound currently being synthesized is interrupted immediately. Conversion of the new text into speech is then begun. If you pass a zero-length string (or, in C, a `null` pointer) to `SpeakString`, the Speech Manager stops any speech previously being synthesized by `SpeakString` without generating additional speech. If your application uses `SpeakString`, it is often a good idea to stop any speech in progress whenever your application receives a suspend event. (Note, however, that calling `SpeakString` with a zero-length string has no effect on speech channels other than the one managed internally by the Speech Manager for the `SpeakString` function.)

The text passed to the `SpeakString` function may contain embedded speech commands, which are described in the chapter "Speech Manager" in this book.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeakString` function are

Trap macro	Selector
_SoundDispatch	$0220000C

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough memory to speak
synthOpenFailed	–241	Could not open another speech synthesizer channel

SEE ALSO

For an example of how to read a text string using the SpeakString function, see "Generating Speech From a String" on page 46. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of event filter functions.

SpeechBusy

You can use the SpeechBusy function to determine whether any channels of speech are currently synthesizing speech.

```
FUNCTION SpeechBusy: Integer;
```

DESCRIPTION

The SpeechBusy function returns the number of speech channels that are currently synthesizing speech in the application. This is useful when you want to ensure that an earlier speech request has been completed before having the system speak again. Note that paused speech channels are counted among those that are synthesizing speech.

The speech channel that the Speech Manager allocates internally in response to calls to the SpeakString function is counted in the number returned by SpeechBusy. Thus, if you use just SpeakString to initiate speech, SpeechBusy always returns 1 as long as speech is being produced. When SpeechBusy returns 0, all initiated speech has finished.

SPECIAL CONSIDERATIONS

You can call the SpeechBusy function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SpeechBusy function are

Trap macro	Selector
_SoundDispatch	$003C000C

Introduction to Sound on the Macintosh

Sound Manager

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 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 128 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 144.

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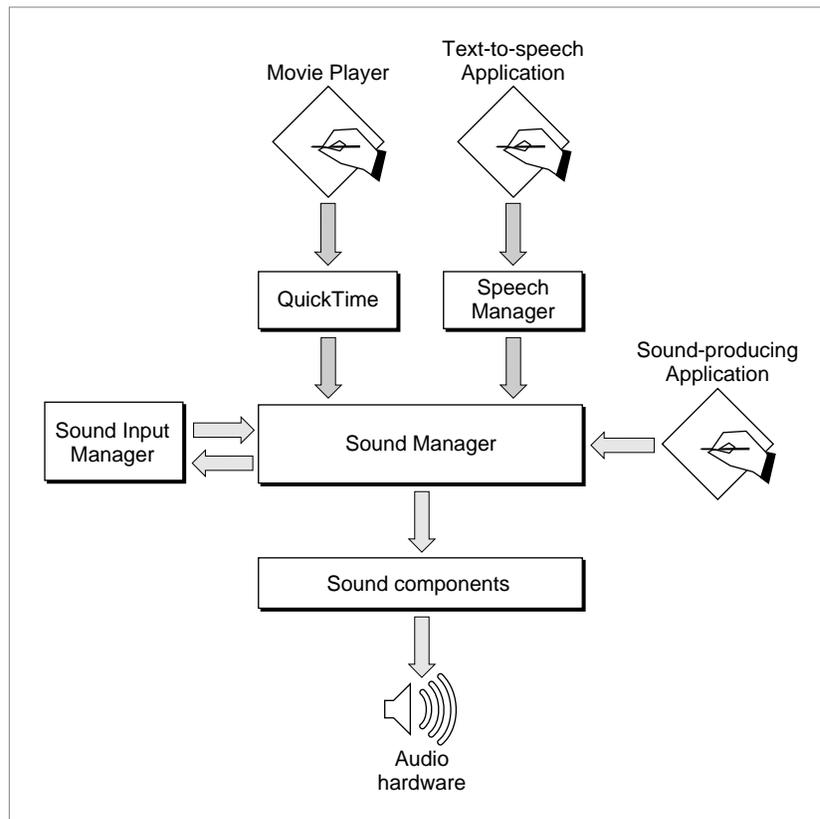
The Sound Manager was first introduced in system software version 6.0 and has been significantly enhanced since that time. Prior to system software version 6.0, applications could create sounds using the Sound Driver.

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
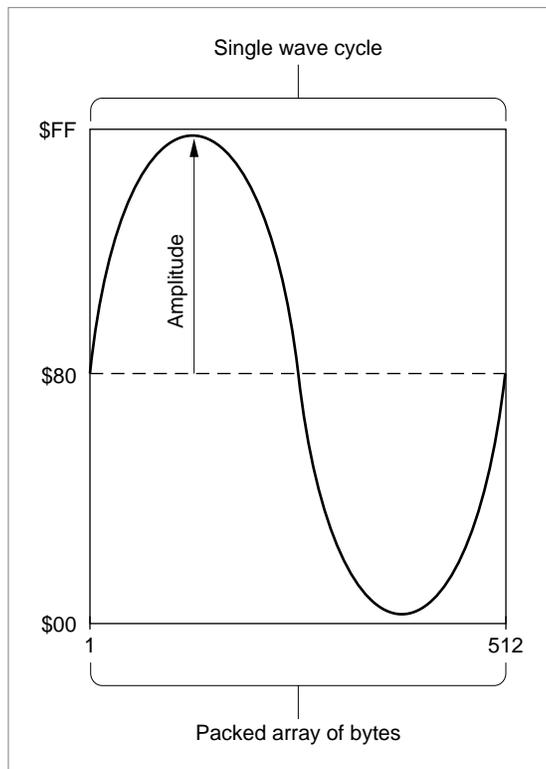
Sampled-Sound Data

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 129, and the structure of AIFF and AIFF-C files is given in "Sound Files" on page 136. 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
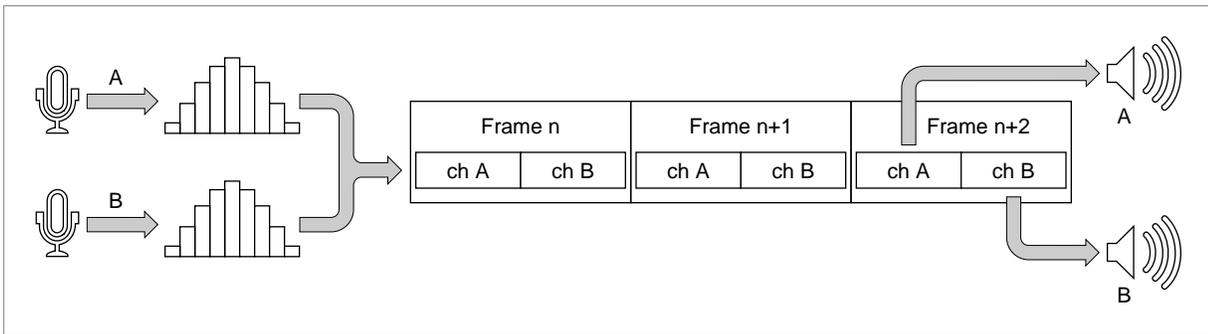
Each **sample point** of noncompressed sound data in a sample frame is, for sound files, a linear, two's complement value, and, for sound resources, a binary offset value. Sample points are from 1 to 32 bits wide. The size is usually 8 bits, but a different size can be specified in the `sampleSize` field 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 129 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 72. Also see "Sound Command Numbers" beginning on page 147 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 90.

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 74 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 87 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 96 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 101 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 111 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 121 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 123 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 101. 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 116.

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 101 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}
    waveInitChannel2    = $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 146 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.

▲ **W AR N I N G**
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 107 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 101.

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 84 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 107 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 107 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 145.

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 155 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 98 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 83 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 112 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 112. 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 74. 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 108 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 75 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 80) 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 92 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 101 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 128 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 98) 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 99.

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 202 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 101.

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 136, 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 90 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 166 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 112. 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
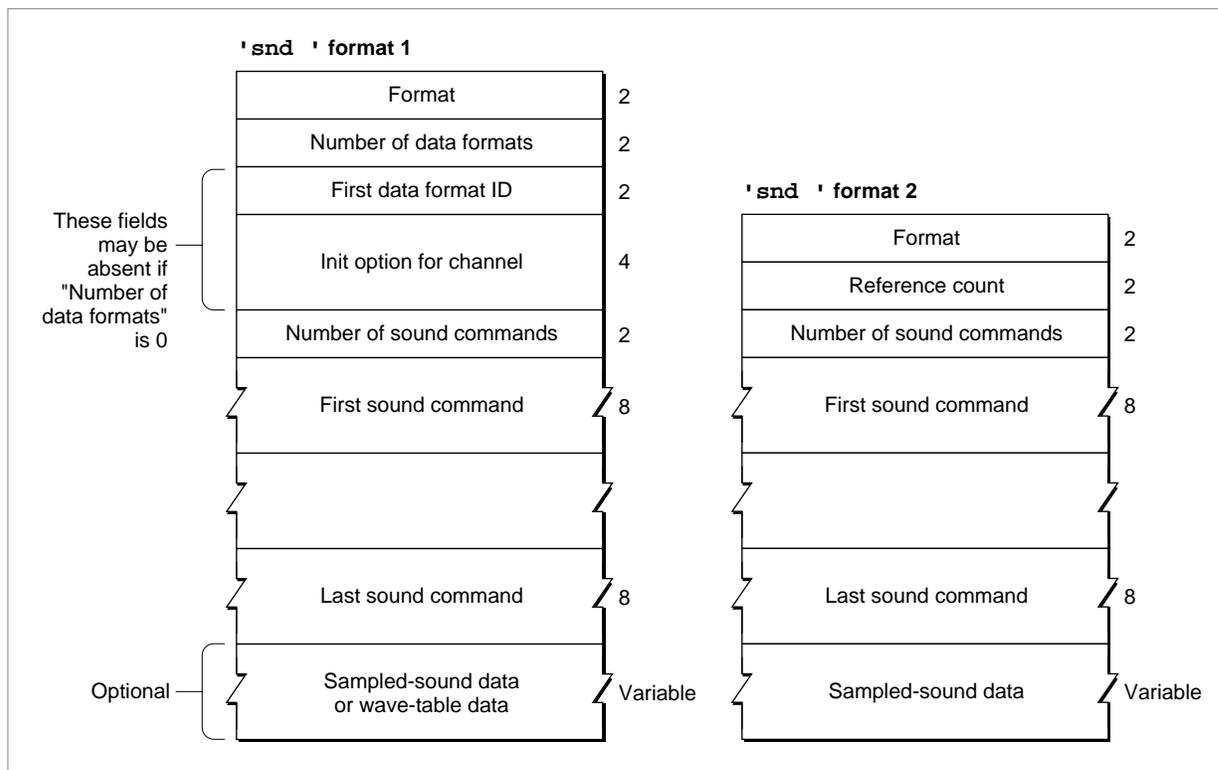
IMPORTANT

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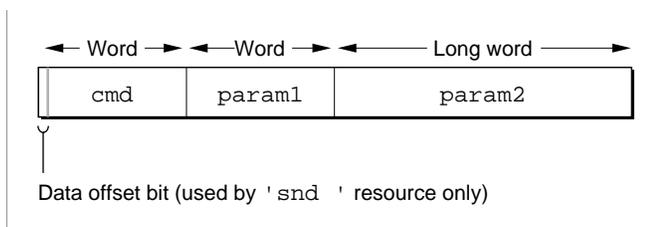
The offset bit is used only by sound commands that are stored in sound resources of type 'snd ' and that have associated sound data (that is, sampled-sound or wave-table data).

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 159.

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 163 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
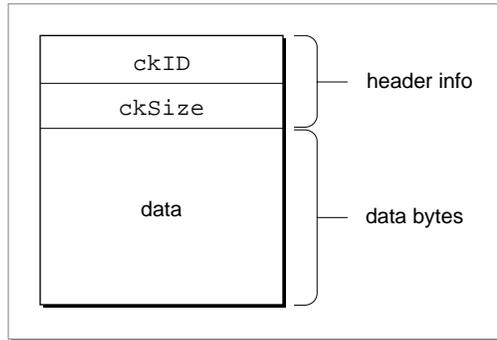
The header information of a chunk has this structure:

```
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 64.)

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 71.

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 64.

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
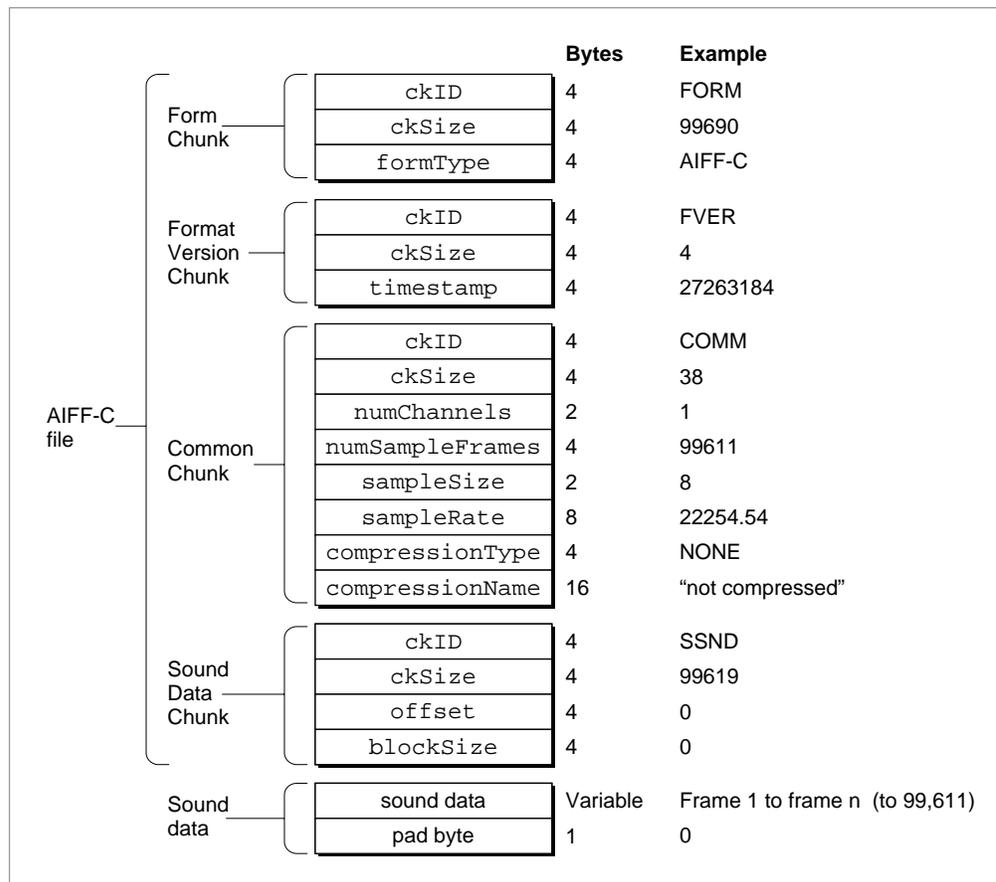
Every Form Chunk must contain a Common Chunk, and every AIFF-C file must contain a Format Version Chunk. In addition, if the sampled sound has a length greater than 0, there must be a Sound Data Chunk in the Form Chunk. All other chunk types are optional. Your application should be able to read all the 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.

The section "Data Structures" beginning on page 154 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 174 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 206 describes callback procedures and completion routines that your application might need to define.

The section "Resources" beginning on page 209 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 182.)

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}
  waveInitChannel2      = $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 154) 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 81 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 147 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 207 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 98 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 136.

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 71. 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 98 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 71. 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 141 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 174 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 174 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 141 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 153 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 136.

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 206 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. ◆

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 192. For information on reading or adjusting the system alert sound volume, see "Controlling Volume Levels" beginning on page 194.

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 75. ▲

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 101 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 116.

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 206.

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.

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 75.

For information on the format of a callback procedure, see "Callback Procedures" on page 207.

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 147 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 97.

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 81.

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 89.

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.

CHAPTER 2

Sound Manager

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 147 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

CHAPTER 2

Sound Manager

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 156.

Sound Manager

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

Sound Manager Reference **191**

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 112 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 112 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 151 for details on these two sound commands; see "Managing Sound Volumes" beginning on page 86 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

Sound Manager

2

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 90.

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 208.

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 123.

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 147.

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 107.

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 102.

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 123.

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 129.

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
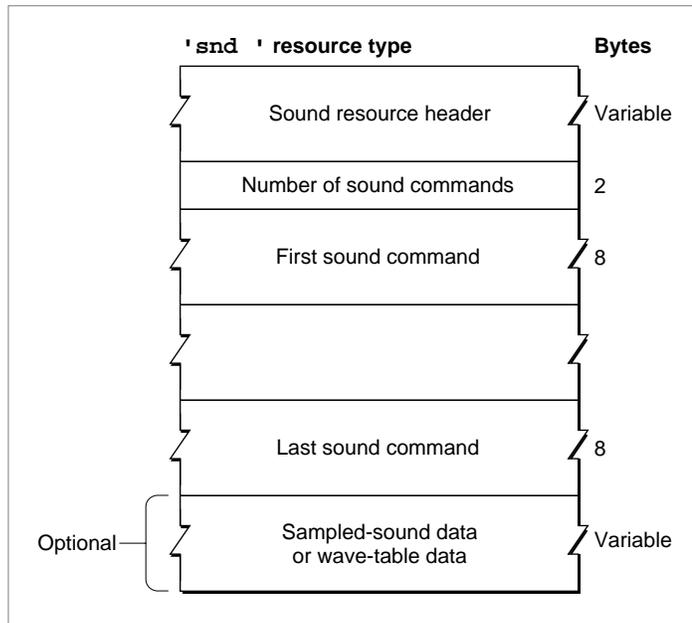
Often, you can create a sound resource simply by using the SndRecord function, documented in the chapter "Introduction to Sound on the Macintosh" in this book. However, you can also define a sound resource manually. This is especially useful for sound resources that are simply series of sound commands 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
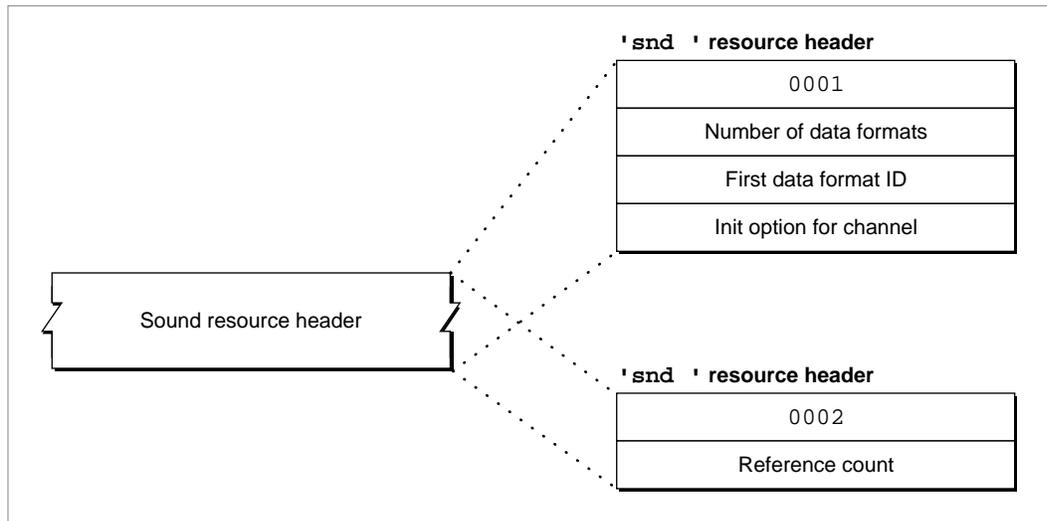
■ Format 1 sound resource header. For format 1 'snd ' resources, the sound resource header includes a word that indicates the number of data types to be sent to the sound channel. Because a sound channel cannot play more than one type of sound data, you should typically specify either $00 or $01 in this 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.

Sound Input Manager

This chapter describes the Sound Input Manager, the part of the Macintosh system software that controls the recording of sound through sound input devices. You can use the Sound Input Manager to display and manage the sound recording dialog box. This ensures that the user is presented with a consistent and standard user interface for sound recording. You can, however, also use Sound Input Manager routines to record sound without the sound recording dialog box or to interact directly with a sound input device driver.

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, and in particular with the portions of that chapter that concern sound recording. That chapter explains how your application can record either a sound resource or a sound file using the standard sound recording dialog box. You need to read this chapter only if you need to interact with the Sound Input Manager at a lower level than is allowed by the high-level functions `SndRecord` and `SndRecordToFile`. For example, you need to read this chapter to learn how to

■ record sound without using the sound recording dialog box

■ interact with a sound input device driver

■ write a sound input device driver

To use this chapter, you should also be familiar with the chapter "Sound Manager" in this book, especially the portions of that chapter that describe

■ the format of sampled-sound data

■ the Macintosh Audio Compression and Expansion (MACE) routines

■ the structure of sound resources and sound files

■ the use of the `Gestalt` function to determine whether certain sound-related facilities are available.

If you are writing a sound input device driver, you should already be familiar with writing device drivers in general, as described in the book *Inside Macintosh: Devices.*

About the Sound Input Manager

The Sound Input Manager uses sound input device drivers to allow applications to access sound input hardware in a device-independent way. A **sound input device driver** is a standard Macintosh device driver used to interface to an audio digitizer or other recording hardware. If you use the Sound Input Manager's high-level routines, the Sound Input Manager handles all communication with a sound input device driver for you. If, however, you need to use the Sound Input Manager's low-level routines, you must open a sound input device driver yourself. You might also need to get information about certain attributes of a sound input device. Sound input device drivers allow your application to query a device about such attributes.

Sound Recording Without the Standard Interface

The Sound Input Manager provides your application with the ability to record and digitally store sounds in a device-independent manner even if your application does not use the standard sound recording interface. In cases where you need very fine control over the recording process, you can call various low-level sound input routines.

Your application can obtain control over sound recording in two different ways. First, if your application uses the sound recording dialog box, you can modify the dialog box's features by defining a custom filter procedure, as explained in detail in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.* Second, if your application needs to fine tune the sound recording process itself (or if your application does not use the standard sound recording dialog box), then the application must use the Sound Input Manager's low-level routines.

In instances where you need to gain greater control over the recording process, you can use a set of routines that manipulate the incoming sound data by using sound parameter blocks. The parameter blocks contain information about the current recording device, the length recorded, a routine to call on completion of the recording, and so forth. You can call the `SPBRecord` function (or the `SPBRecordToFile` function) to begin a recording. Then you can use the functions `SPBPauseRecording`, `SPBResumeRecording`, and `SPBStopRecording` to control the recording. Note that you need to open a device (using the `SPBOpenDevice` function) before you can record from it. On completion of the recording, you should close the device (using the `SPBCloseDevice` function).

If you do record sounds using the Sound Input Manager's low-level routines, you also need to set up your own sound resource headers or sound files, because the Sound Input Manager's low-level routines return raw sampled-sound data to your application. The Sound Input Manager provides two functions, `SetupSndHeader` and `SetupAIFFHeader`, that allow you to set up your own sound resource headers or sound files.

Interaction With Sound Input Devices

The Sound Input Manager provides routines that allow your application to request information about a sound input device or to change a sound input device's settings. The types of information you can obtain about a sound input device include

■ the name, icon, and icon mask of the device driver

■ whether the device driver supports asynchronous recording

■ the device's settings, such as the number of channels the device is to record, the compression type, the number of bytes per sample at the current compression setting, and the sample rate to be produced by the device

■ the range of compression types, sample rates, and sample sizes that the device supports

You can also use the Sound Input Manager to change some of a sound input device's settings and to turn features on and off. For example, you can turn on and off automatic gain control on some device drivers. **Automatic gain control** moderates sound recording to give a consistent signal level. Second, you can turn on and off the **playthrough feature,** which allows the user to hear through the Macintosh speaker the sound being recorded. Third, you can turn on and off **VOX recording,** or voice-activated recording, which allows your application to record only when the amplitude of sound input exceeds a certain level. You can use VOX recording either to prevent recording from starting until sound is at least a certain amplitude or to automatically stop recording when sound falls below a certain amplitude. This latter capability is called **VOX stopping**.

An important feature of sound input devices is **continuous recording.** All sound input devices that support asynchronous recording should support continuous recording as well. Continuous recording allows your application to make several consecutive calls to the `SPBRecord` function without losing data between calls. For example, you might need to record a lengthy sound to disk but not be able to fit the entire sound into RAM. Thus, it's important to be able to save a buffer of data to disk while the sound input device driver continues to collect recorded data. The Sound Input Manager's `SndRecordToFile` function relies on continuous recording.

To get information about a device or to turn features on and off, you can use the `SPBGetDeviceInfo` and `SPBSetDeviceInfo` functions. These functions allow you to use sound input device information selectors to specify what type of information you need to know about the device or what settings you wish to change.

Sound Input Device Drivers

The Sound Input Manager also provides several routines intended for use only by sound input device drivers. Sound input device drivers need to register themselves with the Sound Input Manager by calling the `SPBSignInDevice` function. This makes that device visible in the Sound In control panel for possible selection as the current input device. You can remove a device from that panel by calling the `SPBSignOutDevice` function.

Sound Input Manager 3

For Macintosh computers with built-in sound recording hardware, the system software includes a sound input device driver. This driver automatically calls `SPBSignInDevice` when the computer starts up. If you are creating a sound input device driver for some other sound recording hardware, your device driver must register itself at startup time. Once your driver is registered, it must respond to Status, Control, and Read calls issued by the Sound Input Manager. The Sound Input Manager issues Status calls to get information about a device, Control calls to set device settings, and Read calls to initiate recording.

Using the Sound Input Manager

You can use the Sound Input Manager to record sounds with the sound recording dialog box, to record sounds directly from a device, to get and set information about a sound input device, and to register your sound input device driver so that it can respond to Read, Status, and Control calls. This section does not explain how to record sounds using the sound recording dialog box; for information on that, see the chapter "Introduction to Sound on the Macintosh" in this book.

Recording Sounds Directly From a Device

The Sound Input Manager provides a number of routines that you can use for low-level control over the recording process (such as the ability to intercept sound input data at interrupt time). You can open a sound input device and read data from it by calling these low-level Sound Input Manager routines. Several of those routines access information through a **sound input parameter block,** which is defined by the `SPB` data type:

```
TYPE SPB =
RECORD
   inRefNum:            LongInt;    {reference number of input device}
   count:               LongInt;    {number of bytes to record}
   milliseconds:        LongInt;    {number of milliseconds to record}
   bufferLength:        LongInt;    {length of buffer to record into}
   bufferPtr:           Ptr;        {pointer to buffer to record into}
   completionRoutine:   ProcPtr;    {pointer to a completion routine}
   interruptRoutine:    ProcPtr;    {pointer to an interrupt routine}
   userLong:            LongInt;    {for application's use}
   error:               OSErr;      {error returned after recording}
   unused1:             LongInt;    {reserved}
END;
```

The `inRefNum` field indicates the reference number of the sound input device from which the recording is to occur. You can obtain the reference number of the default sound input device by using the `SPBOpenDevice` function.

The count, milliseconds, and bufferLength fields jointly determine the length of recording. The count field indicates the number of bytes to record; the milliseconds field indicates the number of milliseconds to record; and the bufferLength field indicates the length in bytes of the buffer into which the recorded sound data is to be placed. If the count and milliseconds fields are not equivalent, then the field which specifies the longer recording time is used. If the buffer specified by the bufferLength field is shorter than this recording time, then the recording time is truncated so that the recorded data can fit into the buffer specified by the bufferPtr field. The Sound Input Manager provides two functions, SPBMilliSecondsToBytes and SPBBytesToMilliSeconds, that allow you to convert between byte and millisecond values.

After recording finishes, the count and milliseconds fields indicate the number of bytes and milliseconds actually recorded.

The completionRoutine and interruptRoutine fields allow your application to define a sound input completion routine and a sound input interrupt routine, respectively. More information on these routines is provided later in this section.

The userLong field contains a long integer that is provided for your application's own use. You can use this field, for instance, to pass a handle to an application-defined structure to the sound input completion or interrupt routine. Or, you can use this field to store the value of your application's A5 register, so that your sound input completion or interrupt routine can access your application's global variables. For more information on preserving the value of the A5 register, see the discussion of the SetA5 and SetCurrentA5 functions in the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.*

The error field describes any errors that occur during the recording. This field contains a value greater than 0 while recording unless an error occurs, in which case it contains a value less than 0 that indicates an operating system error. Your application can poll this field to check on the status of an asynchronous recording. If recording terminates without an error, this field contains 0.

Listing 3-1 shows how to set up a sound parameter block and record synchronously using the SPBRecord function. This procedure takes one parameter, a handle to a block of memory in which the recorded sound data is to be stored. It is assumed that the block of memory is large enough to hold the sound to be recorded.

Listing 3-1   Recording directly from a sound input device

```
PROCEDURE MyRecordSnd (mySndH: Handle);
CONST
   kAsync = TRUE;
   kMiddleC = 60;
VAR
   mySPB:        SPB;        {a sound input parameter block}
   myInRefNum:   LongInt;    {device reference number}
```

```
   myBuffSize:    LongInt;    {size of buffer to record into}
   myHeadrLen:    Integer;    {length of sound header}
   myNumChans:    Integer;    {number of channels}
   mySampSize:    Integer;    {size of a sample}
   mySampRate:    Fixed;      {sample rate}
   myCompType:    OSType;     {compression type}
   myErr:         OSErr;
BEGIN
   {Open the default input device for reading and writing.}
   myErr := SPBOpenDevice('', siWritePermission, myInRefNum);

   IF myErr = noErr THEN
   BEGIN
      {Get current settings of sound input device.}
      MyGetDeviceSettings(myInRefNum, myNumChans, mySampRate,
                          mySampSize, myCompType);

      {Set up handle to contain the 'snd ' resource header.}
      myErr := SetupSndHeader(mySndH, myNumChans, mySampRate,mySampSize,
                                 myCompType, kMiddleC, 0, myHeadrLen);

      {Leave room in buffer for the sound resource header.}
      myBuffSize := GetHandleSize(mySndH) - myHeadrLen;

      {Lock down the sound handle until the recording is over.}
      HLockHi(mySndH);

      {Set up the sound input parameter block.}
      WITH mySPB do
      BEGIN
         inRefNum := myInRefNum;            {input device reference number}
         count := myBuffSize;               {number of bytes to record}
         milliseconds := 0;                 {no milliseconds}
         bufferLength := myBuffSize;        {length of buffer}
         bufferPtr := Ptr(ORD4(mySndH^) + myHeadrLen);
                                            {put data after 'snd ' header}
         completionRoutine := NIL;          {no completion routine}
         interruptRoutine := NIL;           {no interrupt routine}
         userLong := 0;                     {no user data}
         error := noErr;                    {clear error field}
         unused1 := 0;                      {clear reserved field}
      END;
```

```
      {Record synchronously through the open sound input device.}
      myErr := SPBRecord(@mySPB, NOT kAsync);

      HUnlock(mySndH);                        {unlock the handle}

      {Indicate the number of bytes actually recorded.}
      myErr := SetupSndHeader(mySndH, myNumChans, mySampRate, mySampSize,
                              myCompType, kMiddleC, mySPB.count,
                              myHeadrLen);

      {Close the input device.}
      myErr := SPBCloseDevice(myInRefNum);
   END;
END;
```

The MyRecordSnd procedure defined in Listing 3-1 opens the default sound input device by using the SPBOpenDevice function. You can specify one of two values for the permission parameter of SPBOpenDevice:

```
CONST
   siReadPermission  = 0;  {open device for reading}
   siWritePermission = 1;  {open device for reading/writing}
```

You must open a device for both reading and writing if you intend to use the SPBSetDeviceInfo function or the SPBRecord function. If SPBOpenDevice successfully opens the specified device for reading and writing, MyRecordSnd calls the MyGetDeviceSettings procedure (defined in Listing 3-3 on page 222). That procedure calls the Sound Input Manager function SPBGetDeviceInfo (explained in "Getting and Setting Sound Input Device Information" on page 220) to determine the current number of channels, sample rate, sample size, and compression type in use by the device.

This information is then passed to the SetupSndHeader function, which sets up the handle mySndH with a sound header describing the current device settings. After doing this, MyRecordSnd sets up a sound input parameter block and calls the SPBRecord function to record a sound. Note that the handle must be locked during the recording because the parameter block contains a pointer to the input buffer. After the recording is done, MyRecordSnd once again calls the SetupSndHeader function to fill in the actual number of bytes recorded.

If the MyRecordSnd procedure defined in Listing 3-1 executes successfully, the handle mySndH points to a resource of type 'snd '. Your application can then synchronously play the recorded sound, for example, by executing the following line of code:

```
myErr := SndPlay(NIL, mySndH, FALSE);
```

For more information on playing sounds your application has recorded, see the chapter "Sound Manager" in this book.

Defining a Sound Input Completion Routine

The `completionRoutine` field of the sound parameter block record contains the address of a completion routine that is executed when the recording terminates normally, either by reaching its prescribed time or size limits or by the application calling the `SPBStopRecording` function. A completion routine should have the following format:

```
PROCEDURE MySICompletionRoutine (inParamPtr: SPBPtr);
```

The completion routine is passed the address of the sound input parameter block that was passed to the `SPBRecord` function. You can gain access to other data structures in your application by passing an address in the `userLong` field of the parameter block. After the completion routine executes, your application should check the `error` field of the sound input parameter block to see if an error code was returned.

Your sound input interrupt routine is always called at interrupt time, so it should not call routines that might allocate or move memory or assume that A5 is set up. For more information on sound input interrupt routines, see "Sound Input Interrupt Routines" beginning on page 265.

Defining a Sound Input Interrupt Routine

The `interruptRoutine` field of the sound input parameter block contains the address of a routine that executes when the internal buffers of an asynchronous recording device are filled. The internal buffers contain raw sound samples taken directly from the input device. The interrupt routine can modify the samples in the buffer in any way it requires. The processed samples are then written to the application buffer. If compression is enabled, the modified data is compressed after your interrupt routine operates on the samples and before the samples are written to the application buffer.

Your sound input interrupt routine is always called at interrupt time, so it should not call routines that might allocate or move memory or assume that A5 is set up. For more information on sound input interrupt routines, see "Sound Input Interrupt Routines" beginning on page 265.

Getting and Setting Sound Input Device Information

You can get information about a specific sound input device and alter a sound input device's settings by calling the functions `SPBGetDeviceInfo` and `SPBSetDeviceInfo`. These functions accept **sound input device information selectors** that determine which information you need or want to change. The selectors currently available are defined by constants of type `OSType`.

Here is a list of the selectors that all sound input device drivers must support. For complete details on all the selectors described in this section, see "Sound Input Device Information Selectors" beginning on page 229.

```
CONST
  siAsync                = 'asyn';   {asynchronous capability}
  siChannelAvailable     = 'chav';   {number of channels available}
  siCompressionAvailable = 'cmav';   {compression types available}
  siCompressionFactor    = 'cmfa';   {current compression factor}
  siCompressionType      = 'comp';   {compression type}
  siContinuous           = 'cont';   {continuous recording}
  siDeviceBufferInfo     = 'dbin';   {size of interrupt buffer}
  siDeviceConnected      = 'dcon';   {input device connection status}
  siDeviceIcon           = 'icon';   {input device icon}
  siDeviceName           = 'name';   {input device name}
  siLevelMeterOnOff      = 'lmet';   {level meter state}
  siNumberChannels       = 'chan';   {current number of channels}
  siRecordingQuality     = 'qual';   {recording quality}
  siSampleRate           = 'srat';   {current sample rate}
  siSampleRateAvailable  = 'srav';   {sample rates available}
  siSampleSizeAvailable  = 'ssav';   {sample sizes available}
  siSampleSize           = 'ssiz';   {current sample size}
  siTwosComplementOnOff  = 'twos';   {two's complement state}
```

The Sound Input Manager defines several selectors that specifically help it interact with sound input device drivers. Your application should not use any of these selectors, but if you are implementing a sound input device driver, you need to support these selectors. They are:

```
CONST
  siCloseDriver          = 'clos';   {release driver}
  siInitializeDriver     = 'init';   {initialize driver}
  siPauseRecording       = 'paus';   {pause recording}
  siUserInterruptProc    = 'user';   {set sound input interrupt routine}
```

Finally, there are a number of sound input device information selectors that sound input device drivers can optionally support. If you are writing an application, you can use these selectors to interact with a sound input device driver, but you should be aware that some drivers might not support all of them. To determine if a driver supports one of these selectors, you can use the SPBGetDeviceInfo function. If no errors are returned, then the selector is supported when using the SPBGetDeviceInfo and the SPBSetDeviceInfo functions.

```
CONST
  siActiveChannels       = 'chac';   {channels active}
  siActiveLevels         = 'lmac';   {levels active}
  siAGCOnOff             = 'agc ';   {automatic gain control state}
  siCompressionHeader    = 'cmhd';   {get compression header}
  siCompressionNames     = 'cnam';   {return compression type names}
```

```
siInputGain              = 'gain';   {input gain level}
siInputSource            = 'sour';   {input source selector}
siInputSourceNames       = 'snam';   {input source names}
siOptionsDialog          = 'optd';   {display options dialog box}
siPlayThruOnOff          = 'plth';   {play-through state}
siStereoInputGain        = 'sgai';   {stereo input gain level}
siVoxRecordInfo          = 'voxr';   {VOX record parameters}
siVoxStopInfo            = 'voxs';   {VOX stop parameters}
```

The format of the relevant data (either returned by the Sound Input Manager or provided by you) depends on the selector you provide. For example, if you want to determine the name of some sound input device, you can pass to the SPBGetDeviceInfo function the siDeviceName selector and a pointer to a 256-byte buffer. If the SPBGetDeviceInfo function can get the information, it fills that buffer with the name of the specified sound input device. Listing 3-2 illustrates one way you can determine the name of a particular sound input device.

Listing 3-2 Determining the name of a sound input device

```
FUNCTION MyGetDeviceName (myRefNum: LongInt; VAR dName: Str255): OSErr;
BEGIN
   MyGetDeviceName := SPBGetDeviceInfo(myRefNum, siDeviceName, Ptr(@dName));
END;
```

Note

You can get the name and icon of all connected sound input devices without using sound input information selectors by using the SPBGetIndexedDevice function, which is described on page 259. ◆

Some selectors cause the SPBGetDeviceInfo function to return data of other types. Listing 3-3 illustrates how to determine the number of channels, the sample rate, the sample size, and the compression type currently in use by a given sound input device. (The procedure defined in Listing 3-3 is called in the procedure defined in Listing 3-1.)

Listing 3-3 Determining some sound input device settings

```
PROCEDURE MyGetDeviceSettings (myRefNum: LongInt;
                               VAR numChannels: Integer;
                               VAR sampleRate: Fixed;
                               VAR sampleSize: Integer;
                               VAR compressionType: OSType);
VAR
   myErr:      OSErr;
BEGIN
   {Get number of active channels.}
```

```
    myErr := SPBGetDeviceInfo (myRefNum, siNumberChannels, Ptr(@numChannels));
    {Get sample rate.}
    myErr := SPBGetDeviceInfo(myRefNum, siSampleRate, Ptr(@sampleRate));
    {Get sample size.}
    myErr := SPBGetDeviceInfo(myRefNum, siSampleSize, Ptr(@sampleSize));
    {Get compression type.}
    myErr := SPBGetDeviceInfo(myRefNum, siCompressionType,
                                              Ptr(@compressionType));
END;
```

All of the selectors that return a handle allocate the memory for that handle in the current heap zone; you are responsible for disposing of that handle when you are done with it, and you should verify that there is enough memory for such a handle before calling the selector.

Writing a Sound Input Device Driver

This section describes what you need to do when you do write a sound input device driver. If you write a sound input device driver, you should set the `drvrFlags` field of the sound input device driver's header to indicate that the driver can handle Status, Control, and Read requests. The driver header should also indicate that the driver needs to be locked.

IMPORTANT

You don't need to write a device driver to use sound input capabilities. ▲

After you create a device driver, you must write an extension that installs it. Before your extension installs the driver, it should pass the `Gestalt` function the `gestaltSoundAttr` attribute selector and inspect the `gestaltSoundIOMgrPresent` bit to determine if the sound input routines are available. If so, the extension should install the sound input device driver into the unit table just as any other driver must be installed.

After installing the driver, the extension must then make an Open request to the driver, so that the driver can perform any necessary initialization. In particular, the driver might set the `dCtlStorage` field of the device control entry to a pointer or a handle to a block in the system heap containing all of the variables that it might need. Finally, the device driver signs into the Sound Input Manager by calling the `SPBSignInDevice` function.

Once signed in, a driver can receive Status, Control, and Read requests from the Sound Input Manager. On entry, the A0 register contains a pointer to a standard Device Manager parameter block, and the A1 register contains a pointer to the device control entry. For more information on using registers in a device driver, see *Inside Macintosh: Devices*.

Responding to Status and Control Requests

The Sound Input Manager supports sound input device information selectors by sending your device driver Status and Control requests. It uses Status requests to get information about your device; it uses Control requests to change settings of your sound input device.

The behavior of your sound input device driver in response to Status and Control requests depends on the value of the csCode field of the Device Manager control parameter block. If the csCode field contains 2, then the sound input information selector is passed in the first 4 bytes of the csParam field of the Device Manager control parameter block. For Status requests, the next 18 bytes can be used for your device driver to pass information back to an application. For Control requests, these 18 bytes are used by an application to pass data to your sound input device driver.

Figure 3-1 shows the contents of the csParam field of the Device Manager control parameter block for a sample Status request. The first four bytes of the csParam field contain the input selector 'srav', which is a request for the available sample rates. The next four bytes of the field contain a pointer to an application-supplied buffer in which to return the data (the number of rates available) from the Status request.

Figure 3-1   An example of the csParam field for a Status request
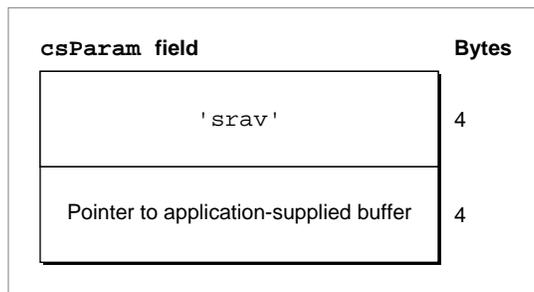
On exit from the Status request, your sound input device driver can respond in one of two ways. If you are returning fewer than 18 bytes of data, your device driver should specify in the first 4 bytes of the csParam field of the Device Manager control parameter block the number of bytes of data being returned and place the data in the following 18 bytes. In this case, the Sound Input Manager copies the data to the application-supplied buffer identified in Figure 3-1. If you are returning more than 18 bytes of data, your device driver should copy the data to the application-supplied buffer. In this case, your device driver needs to place a zero in the first 4 bytes of the csParam field to indicate to the Sound Input Manager that the data has already been copied to the application-supplied buffer.

Figure 3-2 shows the contents of the csParam field of the Device Manager control parameter block for a sample Control request. The first four bytes of the csParam field contain the input selector 'srat' which determines the sample rate for the sound input

device. The next eighteen bytes contain the data, which in this example is the sample rate to set for your sound input device. This is a `Fixed` value of four bytes in length.

Figure 3-2   An example of the `csParam` field for a Control request
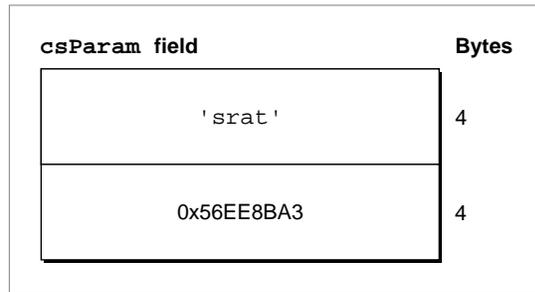
Note

Some sound input information selectors require your sound input device driver to allocate a handle in which to store information. In this case, your driver should attempt to allocate an appropriately sized handle in the current heap zone. If allocation fails, your driver should return the appropriate Memory Manager result code. ◆

Your sound input device driver must respond to a core set of selectors, but the remaining selectors defined by Apple are optional. Your device driver might also define private selectors to support proprietary features. (Selectors containing all lowercase letters, however, are reserved by Apple.) The section "Getting and Setting Sound Input Device Information" beginning on page 220 lists the core selectors and other selectors that have been defined.

If the `csCode` field contains 1 (which can occur only for Control requests), the Sound Input Manager is attempting to stop asynchronous recording; that is, it is issuing a `KillIO` request. In response to this, the driver should stop copying data to the application buffer, update the `ioActCount` field of the request parameter block, and return via an `RTS` instruction.

Before exiting after a Status and Control request, your sound input device driver should fill the D0 register with the appropriate result code or `noErr`. To exit, your sound input device driver should check whether the Status and Control request was executed immediately or was queued.

Note

In current versions of system software, the Sound Input Manager always issues Status and Control requests immediately. This might change in future versions of system software. ◆

Your sound input device driver can determine whether a request is issued immediately by checking the `noQueueBit` in the `ioTrap` field of the Device Manager control parameter block. If the request was made immediately, the Control routine should return

via an `RTS` instruction; if the request was queued, the Control routine should jump to the Device Manager's `IODone` function via the global jump vector `JIODone`. You need to make sure that the A0 and A1 registers are set the same as they are on entry to the device driver or `JIODone` will fail.

Responding to Read Requests

When a sound input device receives a Read request, it must start recording and saving recorded data into the buffer specified by the `ioBuffer` field of the request parameter block. If that field is `NIL`, the driver should record but not save the data. During a Read request, your sound input device driver can access the sound parameter block that initiated recording through the `ioMisc` field of the request parameter block.

If a previous Control request has assigned a sound input interrupt routine to the device driver and your driver records asynchronously, then the driver must call the routine each time its internal buffer becomes filled, setting up registers as described in "Defining a Sound Input Interrupt Routine" on page 220. The buffer size that your device driver specifies in the D1 register should indicate how much your device records during every interrupt. For example, a sound input device driver that uses the serial port might use a buffer as small as 3 bytes. For the built-in sound input port on the Macintosh LC and other Macintosh models, the buffer is 512 bytes long.

Your device driver should update the `ioActCount` field of the request parameter block with the actual number of bytes of sampled-sound data recorded. This allows the Sound Input Manager to monitor the activity of your device driver. Whether your device driver operates synchronously or asynchronously, it should complete recording by jumping to the Device Manager's `IODone` function via the global jump vector `JIODone`. You need to set the D0 register to the appropriate result code before jumping to the Device Manager's `IODone` function.

Supporting Stereo Recording

Many sound input devices support recording stereo sounds (that is, sounds from two or more channels). If you are writing a device driver for a stereo device, you need to make sure that you support the `siNumberChannels`, `siActiveChannels`, and `siActiveLevels` selectors.

The `siNumberChannels` selector controls the number of sound input channels and thereby determines the format of the data stream your device driver produces. If the number of channels is 1, the driver should produce monophonic data in response to a Read request. If the number of channels is 2, the driver should produce interleaved stereo data in response to a Read request.

The `siActiveChannels` selector controls which of the available input channels are used for recording. The active channels are specified using a bitmap value. For example, the value $01 indicates that the first channel (the left channel) is to be used. The value $02 indicates that the second channel (the right channel) is to be used.

The `siNumberChannels` and `siActiveChannels` selectors together determine the exact format of the output data stream. If the current number of channels is 1 and the

current active channel bitmap is $01, the driver should produce a stream of monophonic data containing samples only from the left input channel. If the current number of channels is 1 and the current active channel bitmap is $02, the driver should produce a stream of monophonic data containing samples only from the right input channel. If the current number of channels is 1 and the current active channel bitmap is $03, the driver should mix the right and left channels to produce a stream of monophonic data. If the current number of channels is 2 and the current active channel bitmap is $03, the driver should produce a stream of interleaved samples from the left and right input channels.

Note

If the `siActiveChannels` selector is never passed to a sound input device driver, it's recommended that the active channel default bitmap for both monophonic and stereo recording should be $03. When the active channel bitmap conflicts with the number of channels (for example, there are two channels but the active channel bitmap is $01), you should use the default value of $03. ◆

Supporting Continuous Recording

If your sound input device driver supports continuous recording, it must do more than respond to Status, Control, and Read requests. It must also, if continuous recording is on, begin recording into an internal ring buffer as soon as a Read request completes. The buffer should be made large enough so that the sound input device driver can support successive requests to the `SPBRecord` function in most circumstances; however, if your driver exhausts the internal buffer, your driver should begin recording again at the start of the buffer.

When the sound input device driver receives a subsequent Read request, it should record to the application's buffer first all of the data in the internal ring buffer and then as much fresh data as it can record during one interrupt.

If a Read terminates due to a `KillIO` request, your sound input device driver does not need to continue recording samples to the internal ring buffer until after the next uninterrupted Read request.

Sound Input Manager Reference

This section describes the constants, data structure, and the routines provided by the Sound Input Manager.

Constants

This section describes the constants you can use with the `SPBSetDeviceInfo` and `SPBGetDeviceInfo` functions to set or get device information. It also lists the `Gestalt` function sound attributes selector and the returned bit numbers that are relevant to the Sound Input Manager. All other constants defined by the Sound Input Manager are

described at the appropriate location in this chapter. (For example, the constants that you can use to specify sound recording qualities are described in connection with the `SndRecord` function beginning on page 238.)

Gestalt Selector and Response Bits

You can pass the `gestaltSoundAttr` selector to the `Gestalt` function to determine information about the sound input 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 relevant to the Sound Input Manager are defined by constants:

```
CONST
   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}
```

Constant descriptions

`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.

Note

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

Sound Input Device Information Selectors

You can call the `SPBSetDeviceInfo` and `SPBGetDeviceInfo` functions to set or get information about a sound input device. You pass each of those functions a sound input device information selector in the `infoType` parameter to specify the type of information you need. The available device information selectors are defined by constants.

IMPORTANT

Some of these selectors are intended for use only by the Sound Input Manager and other parts of the system software that need to interact directly with sound input device drivers. (For example, the Sound Input Manager sends the `siCloseDriver` selector to a sound input device driver when it is closing the device.) In general, applications should not use these reserved selectors. ▲

```
CONST
   siActiveChannels     = 'chac';   {channels active}
   siActiveLevels       = 'lmac';   {levels active}
   siAGCOnOff           = 'agc ';   {automatic gain control state}
   siAsync              = 'asyn';   {asynchronous capability}
   siChannelAvailable   = 'chav';   {number of channels available}
   siCloseDriver        = 'clos';   {reserved for internal use only}
   siCompressionAvailable = 'cmav'; {compression types available}
   siCompressionFactor  = 'cmfa';   {current compression factor}
   siCompressionHeader  = 'cmhd';   {return compression header}
   siCompressionNames   = 'cnam';   {return compression type names}
   siCompressionType    = 'comp';   {current compression type}
   siContinuous         = 'cont';   {continuous recording}
   siDeviceBufferInfo   = 'dbin';   {size of interrupt buffer}
   siDeviceConnected    = 'dcon';   {input device connection status}
   siDeviceIcon         = 'icon';   {input device icon}
   siDeviceName         = 'name';   {input device name}
   siInitializeDriver   = 'init';   {reserved for internal use only}
   siInputGain          = 'gain';   {input gain level}
   siInputSource        = 'sour';   {input source selector}
   siInputSourceNames   = 'snam';   {input source names}
   siLevelMeterOnOff    = 'lmet';   {level meter state}
   siNumberChannels     = 'chan';   {current number of channels}
   siOptionsDialog      = 'optd';   {display options dialog box}
   siPauseRecording     = 'paus';   {reserved for internal use only}
```

```
siPlayThruOnOff          = 'plth';    {play-through state}
siRecordingQuality       = 'qual';    {recording quality}
siSampleRate             = 'srat';    {current sample rate}
siSampleRateAvailable    = 'srav';    {sample rates available}
siSampleSize             = 'ssiz';    {current sample size}
siSampleSizeAvailable    = 'ssav';    {sample sizes available}
siStereoInputGain        = 'sgai';    {stereo input gain level}
siTwosComplementOnOff    = 'twos';    {two's complement state}
siUserInterruptProc      = 'user';    {reserved for internal use only}
siVoxRecordInfo          = 'voxr';    {VOX record parameters}
siVoxStopInfo            = 'voxs';    {VOX stop parameters}
```

Constant descriptions

siActiveChannels

Get or set the channels to record from. When setting the active channels, the data passed in is a long integer that is interpreted as a bitmap describing the channels to record from. For example, if bit 0 is set, then the first channel is made active. The samples for each active channel are interleaved in the application's buffer. When reading the active channels, the data returned is a bitmap of the active channels.

siActiveLevels

Get the current signal level for each active channel. The infoData parameter points to an array of integers, the size of which depends on the number of active channels. You can determine how many channels are active by calling SPBGetDeviceInfo with the siNumberChannels selector.

siAGCOnOff   Get or set the current state of the automatic gain control feature. The infoData parameter points to an integer, which is 0 if gain control is off and 1 if it is on.

siAsync   Determine whether the driver supports asynchronous recording functions. The infoData parameter points to an integer, which is 0 if the driver supports synchronous calls only and 1 otherwise. Some sound input drivers do not support asynchronous recording at all, and some might support asynchronous recording only on certain hardware configurations.

siChannelAvailable

Get the maximum number of channels this device can record. The infoData parameter points to an integer, which is the number of available channels.

siCloseDriver   The Sound Input Manager sends this selector when it closes a device previously opened with write permission. The sound input device driver should stop any recording in progress, deallocate the input hardware, and initialize local variables to default settings. Your application should never issue this selector directly. The infoData parameter is unused with this selector.

siCompressionAvailable

Get the number and list of compression types this device can produce. The `infoData` parameter points to an integer, which is the number of compression types, followed by a handle. The handle references a list of compression types, each of type `OSType`.

siCompressionFactor

Get the compression factor of the current compression type. For example, the compression factor for MACE 3:1 compression is 3. If a sound input device driver supports only compression type `'NONE'`, the returned compression type is 1. The `infoData` parameter points to an integer, which is the compression factor.

siCompressionHeader

Get a compressed sound header for the current recording settings. Your application passes in a pointer to a compressed sound header and the driver fills it in. Before calling `SPBGetDeviceInfo` with this selector, you should set the `numFrames` field of the compressed sound header to the number of bytes in the sound. When `SPBGetDeviceInfo` returns successfully, that field contains the number of sample frames in the sound. This selector is needed only by drivers that use compression types that are not directly supported by Apple. If you call this selector after recording a sound, your application can get enough information about the sound to play it or save it in a file. The `infoData` parameter points to a compressed sound header.

siCompressionNames

Get a list of names of the compression types supported by the sound input device. In response to a Status call, a sound input device driver returns, in the location specified by the `infoData` parameter, a handle to a block of memory that contains the names of all compression types supported by the driver. It is the driver's responsibility to allocate that block of memory, but it should not release it. The software issuing this selector is responsible for disposing of the handle. As a result, a device driver must detach any resource handles (by calling `DetachResource`) before returning them to the caller. The data in the handle has the same format as an `'STR#'` resource: a two-byte count of the strings in the resource, followed by the strings themselves. The strings should occur in the same order as the compression types returned by the `siCompressionAvailable` selector. If the driver does not support compression, it returns `siUnknownInfoType`. If the driver supports compression but for some reason not all compression types are currently selectable, it returns a list of all available compression types.

siCompressionType

Get or set the compression type. Some devices allow the incoming samples to be compressed before being placed in your application's input buffer. The `infoData` parameter points to a buffer of type `OSType`, which is the compression type.

siContinuous   Get or set the state of continuous recording from this device. If
                      recording is being turned off, the driver stops recording samples to
                      its internal buffer. Only sound input device drivers that support
                      asynchronous recording support continuous recording. The
                      infoData parameter points to an integer, which is the state of
                      continuous recording (0 is off, 1 is on).

siDeviceBufferInfo
                      Get the size of the device's internal buffer. This information can be
                      useful when you want to modify sound input data at interrupt time.
                      Note, however, that if a driver is recording continuously, then the
                      size of the buffer passed to your sound input interrupt routine might
                      be greater than the size this selector returns because data recorded
                      between calls to SPBRecord as well as recorded during calls to
                      SPBRecord will be sent to your interrupt routine. The infoData
                      parameter points to a long integer, which is the size of the device's
                      internal buffer.

siDeviceConnected
                      Get the state of the device connection. The infoData parameter
                      points to an integer, which is one of the following constants:

                      CONST
                          siDeviceIsConnected   = 1;
                          siDeviceNotConnected   = 0;
                          siDontKnowIfConnected   = -1;

                      The siDeviceIsConnected constant indicates that the device is
                      connected and ready. The siDeviceNotConnected constant
                      indicates that the device is not connected. The
                      siDontKnowIfConnected constant indicates that the Sound Input
                      Manager cannot determine whether the device is connected.

siDeviceIcon   Get the device's icon and icon mask. In response to a Status call, a
                      sound input device driver should return, in the location specified by
                      the infoData parameter, a handle to a block of memory that
                      contains the icon and its mask in the format of an 'ICN#' resource.
                      It is the driver's responsibility to allocate that block of memory, but
                      it should not releasee it. The software issuing this selector is
                      responsible for disposing of the handle. As a result, a device driver
                      should detach any resource handles (by calling DetachResource)
                      before returning them to the caller.

siDeviceName   Get the name of the sound input device. Your application must pass
                      a pointer to a buffer that will be filled in with the device's name. The
                      buffer needs to be large enough to hold a Str255 data type.

siInitializeDriver
                      The Sound Input Manager sends this selector when it opens a sound
                      input device with write permission. The sound input device driver
                      initializes local variables and prepares to start recording. If possible,
                      the driver initializes the device to a sampling rate of 22 kHz, a
                      sample size of 8 bits, mono recording, no compression, automatic
                      gain control on, and all other features off. Your application should

never issue this selector directly. The `infoData` parameter is unused with this selector.

siInputGain   Get and set the current sound input gain. If the available hardware allows adjustment of the recording gain, this selector lets you get and set the gain. In response to a Status call, a sound input driver returns the current gain setting. In response to a Control call, a sound input driver sets the gain level used for all subsequent recording to the specified value. The `infoData` parameter points to a 4-byte value of type `Fixed` ranging from 0.5 to 1.5, where 1.5 specifies maximum gain.

siInputSource   Get and set the current sound input source. If the available hardware allows recording from more than one source, this selector lets you get and set the source. In response to a Status call, a sound input driver returns the current source value; if the driver supports only one source, it returns `siUnknownInfoType`. In response to a Control call, a sound input driver sets the source of all subsequent recording to the value passed in. If the value is less than 1 or greater than the number of input sources, the driver returns `paramErr`; if the driver supports only one source, it returns `siUnknownInfoType`. The `infoData` parameter points to an integer, which is the index of the current sound input source.

siInputSourceNames

Get a list of the names of all the sound input sources supported by the sound input device. In response to a Status call, a sound input device driver returns, in the location specified by the `infoData` parameter, a handle to a block of memory that contains the names of all sound sources supported by the driver. It is the driver's responsibility to allocate that block of memory, but it should not release it. The software issuing this selector is responsible for disposing of the handle. As a result, a device driver must detach any resource handles (by calling `DetachResource`) before returning them to the caller. The data in the handle has the same format as an 'STR#' resource: a two-byte count of the strings in the resource, followed by the strings themselves. The strings should occur in the same order as the input sources returned by the `siInputSource` selector. If the driver supports only one source, it returns `siUnknownInfoType`. If the driver supports more than one source but for some reason not all of them are currently selectable, it returns a list of all available input sources.

siLevelMeterOnOff

Get or set the current state of the level meter. For calls to set the level meter, the `infoData` parameter points to an integer that indicates whether the level meter is off (0) or on (1). To get the level meter setting, the `infoData` parameter points to two integers; the first integer indicates the state of the level meter, and the second integer contains the level value of the meter. The level meter setting is an integer that ranges from 0 (no volume) to 255 (full volume).

siNumberChannels

Get or set the number of channels this device is to record. The

infoData parameter points to an integer, which indicates the number of channels. Note that this selector determines the format of the data stream output by the driver. If the number of channels is 1, the driver should output monophonic data in response to a Read call. If the number of channels is 2, the driver should output interleaved stereo data.

siOptionsDialog

Determine whether the driver supports an Options dialog box (SPBGetDeviceInfo) or cause the driver to display the Options dialog box (SPBSetDeviceInfo). This dialog box is designed to allow the user to configure device-specific features of the sound input hardware. With SPBGetDeviceInfo, the infoData parameter points to an integer, which indicates whether the driver supports an Options dialog box (1 if it supports it, 0 otherwise). With SPBSetDeviceInfo, the infoData parameter is unused.

siPauseRecording

The Sound Input Manager uses this selector to get or set the current pause state. The sound input device driver continues recording but does not store the sampled data in a buffer. Your application should never issue this selector directly. The infoData parameter points to an integer, which indicates the state of pausing (0 is off, 1 is on).

siPlayThruOnOff

Get or set the current play-through state and volume. The infoData parameter points to an integer, which indicates the current play-through volume (1 to 7). If that integer is 0, then play-through is off.

siRecordingQuality

Get or set the current quality of recorded sound. The infoData parameter points to a buffer of type OSType, which is the recording quality. Currently three qualities are supported, defined by these constants:

```
CONST
    siBestQuality            = 'best';
    siBetterQuality          = 'betr';
    siGoodQuality            = 'good';
```

These qualities are defined by the sound input device driver. Usually *best* means monaural, 8-bit, 22 kHz, sound with no compression.

siSampleRate   Get or set the sample rate to be produced by this device. The sample rate must be in the range 0 to 65535.65535 Hz. The sample rate is declared as a Fixed data type. In order to accommodate sample rates greater than 32 kHz, the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768. The infoData parameter points to a buffer of type Fixed, which is the sample rate.

siSampleRateAvailable

Get the range of sample rates this device can produce. The

infoData parameter points to an integer, which is the number of sample rates the device supports, followed by a handle. The handle references a list of sample rates, each of type `Fixed`. If the device can record a range of sample rates, the number of sample rates is set to 0 and the handle contains two rates, the minimum and the maximum of the range of sample rates. Otherwise, a list is returned that contains the sample rates supported. In order to accommodate sample rates greater than 32 kHz, the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.

siSampleSize   Get or set the sample size to be produced by this device. Because some compression formats require specific sample sizes, this selector might return an error when compression is used. The `infoData` parameter points to an integer, which is the sample size.

siSampleSizeAvailable
                    Get the range of sample sizes this device can produce. The `infoData` parameter points to an integer, which is the number of sample sizes the device supports, followed by a handle. The handle references a list of sample sizes, each of type `Integer`.

siStereoInputGain
                    Get and set the current stereo sound input gain. If the available hardware allows adjustment of the recording gain, this selector lets you get and set the gain for each of two channels (left or right). In response to a Status call, a sound input driver should return the current gain setting for the specified channel. In response to a Control call, a sound input driver should set the gain level used for all subsequent recording to the specified value. The `infoData` parameter points to two 4-byte values of type `Fixed` ranging from 0.5 to 1.5, where 1.5 specifies maximum gain. The first of these values is equivalent to the gain for the left channel and the second value is equivalent to the gain for the right channel.

siTwosComplementOnOff
                    Get or set the current state of the two's complement feature. This selector only applies to 8-bit data. (16-bit samples are always stored in two's complement format.) If on, the driver stores all samples in the application buffer as two's complement values (that is, –128 to 127). Otherwise, the driver stores the samples as offset binary values (that is, 0 to 255). The `infoData` parameter points to an integer, which is the current state of the two's complement feature (1 if two's complement output is desired, 0 otherwise).

siUserInterruptProc
                    The Sound Input Manager sends this selector to specify the sound input interrupt routine that the sound input device driver should call. Your application should never issue this selector directly. The `infoData` parameter points to a procedure pointer, which is the address of the sound input interrupt routine.

siVoxRecordInfo
                    Get or set the current VOX recording parameters. The `infoData` parameter points to two integers. The first integer indicates whether

Sound Input Manager

3

VOX recording is on or off (0 if off, 1 if on). The second integer indicates the VOX record trigger value. Trigger values range from 0 to 255 (0 is trigger immediately, 255 is trigger only on full volume).

siVoxStopInfo   Get or set the current VOX stopping parameters. The `infoData` parameter points to three integers. The first integer indicates whether VOX stopping is on or off (0 if off, 1 if on). The second integer indicates the VOX stop trigger value. Trigger values range from 0 to 255 (255 is stop immediately, 0 is stop only on total silence). The third integer indicates how many milliseconds the trigger value must be continuously valid for recording to be stopped. Delay values range from 0 to 65,535.

Data Structures

This section describes the sound input parameter block.

Sound Input Parameter Blocks

The `SPBRecord` and `SPBRecordToFile` functions require a pointer to a sound input parameter block that defines characteristics of the recording. If you define a sound input completion routine or a sound input interrupt routine, your routine receives a pointer to a sound input parameter block. If you are using only the Sound Input Manager's high-level `SndRecord` and `SndRecordToFile` functions, the operation of sound input parameter blocks is transparent to your application. A sound input parameter block is defined by the `SPB` data type.

```
TYPE SPB =
RECORD
   inRefNum:          LongInt;    {reference number of input device}
   count:             LongInt;    {number of bytes to record}
   milliseconds:      LongInt;    {number of milliseconds to record}
   bufferLength:      LongInt;    {length of buffer to record into}
   bufferPtr:         Ptr;        {pointer to buffer to record into}
   completionRoutine: ProcPtr;    {pointer to a completion routine}
   interruptRoutine:  ProcPtr;    {pointer to an interrupt routine}
   userLong:          LongInt;    {for application's use}
   error:             OSErr;      {error returned after recording}
   unused1:           LongInt;    {reserved}
END;
```

Field descriptions

inRefNum   The reference number of the sound input device (as received from the `SPBOpenDevice` function) from which the recording is to occur.

count   On input, the number of bytes to record. On output, the number of bytes actually recorded. If this field specifies a longer recording time

than the `milliseconds` field, then the `milliseconds` field is ignored on input.

`milliseconds`   On input, the number of milliseconds to record. On output, the number of milliseconds actually recorded. If this field specifies a longer recording time than the `count` field, then the `count` field is ignored on input.

`bufferLength`   The length of the buffer into which recorded sound data is placed. The recording time specified by the `count` or `milliseconds` field is truncated to fit into this length, if necessary.

`bufferPtr`   A pointer to the buffer into which recorded data is placed. If this field is `NIL`, then the `count`, `milliseconds`, and `bufferLength` fields are ignored and the recording will continue indefinitely until the `SPBStopRecording` function is called. However, the data is not stored anywhere, so setting this field to `NIL` is useful only if you want to do something in a sound input interrupt routine but do not want to save the recorded sound.

`completionRoutine`
A pointer to a completion routine that is called when the recording terminates as a result of your calling the `SPBStopRecording` function or when the limit specified by the `count` or `milliseconds` field is reached. The completion routine executes only if `SPBRecord` is called asynchronously and therefore is called at interrupt time.

`interruptRoutine`
A pointer to a routine that is called by asynchronous recording devices when their internal buffers are full. You can define a sound input interrupt routine to modify uncompressed sound samples before they are placed into the buffer specified in the `bufferPtr` parameter. The interrupt routine executes only if `SPBRecord` is called asynchronously and therefore is called at interrupt time.

`userLong`   A long integer available for the application's own use. You can use this field, for instance, to pass a handle to an application-defined structure to the completion routine or to the interrupt routine.

`error`   On exit, the error that occurred during recording. This field contains a value greater than 0 while recording unless an error occurs, in which case it contains a value less than 0 that indicates an operating system error. Your application can poll this field to check on the status of an asynchronous recording. If recording terminates without an error, this field contains 0.

`unused1`   Reserved for use by Apple. You should always initialize this field to 0.

Sound Input Manager Routines

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

■ record sounds using the sound recording dialog box

■ open and close sound input devices

■ record sounds directly from sound input devices

■ get information about sound input devices and change device settings

■ construct sound resource and file headers

■ register sound input devices with the Sound Input Manager

■ convert recording times between millisecond and byte values

■ obtain information about the version of the Sound Input Manager that is running

The section "Application-Defined Routines" on page 263 describes the format of sound input completion routines and sound input interrupt routines.

Recording Sounds

The Sound Input Manager provides two high-level sound input functions, `SndRecord` and `SndRecordToFile`, for recording sound. These input routines are analogous to the two Sound Manager functions `SndPlay` and `SndStartFilePlay`. By using these high-level routines, you can be assured that your application presents a user interface that is consistent with that displayed by other applications doing sound input. Both `SndRecord` and `SndRecordToFile` attempt to record sound data from the sound input hardware currently selected in the Sound In control panel.

SndRecord

You can use the `SndRecord` function to record sound resources into memory.

```
FUNCTION SndRecord (filterProc: ProcPtr; corner: Point;
                    quality: OSType; VAR sndHandle: Handle):
                    OSErr;
```

`filterProc`
            A pointer to an event filter function that determines how user actions in the sound recording dialog box are filtered (similar to the `filterProc` parameter specified in a call to the `ModalDialog` procedure). By specifying your own filter function, you can override or add to the default actions of the items in the dialog box. If `filterProc` isn't `NIL`, `SndRecord` filters events by calling the function that `filterProc` points to.

`corner`   The horizontal and vertical coordinates of the upper-left corner of the sound recording dialog box (in global coordinates).

`quality`   The desired quality of the recorded sound.

`sndHandle` On entry, a handle to some storage space or `NIL`. On exit, a handle to a valid sound resource (or unchanged, if the call did not execute successfully).

DESCRIPTION

The `SndRecord` function records sound into memory. The recorded data has the structure of a format 1 `'snd '` resource and can later be played using the `SndPlay` function or can be stored as a resource. `SndRecord` displays a sound recording dialog box and is always called synchronously. Controls in the dialog box allow the user to start, stop, pause, and resume sound recording, as well as to play back the recorded sound. The dialog box also lists the remaining recording time and the current microphone sound level.

The `quality` parameter defines the desired quality of the recorded sound. Currently, three values are recognized for the `quality` parameter:

```
CONST
    siBestQuality       = 'best';   {the best quality available}
    siBetterQuality     = 'betr';   {a quality better than good}
    siGoodQuality       = 'good';   {a good quality}
```

The precise meanings of these parameters are defined by the sound input device driver. For Apple-supplied drivers, this parameter determines whether the recorded sound is to be compressed, and if so, whether at a 6:1 or a 3:1 ratio. The quality `siBestQuality` does not compress the sound and provides the best quality output, but at the expense of increased memory use. The quality `siBetterQuality` is suitable for most nonvoice recording, and `siGoodQuality` is suitable for voice recording.

The `sndHandle` parameter is a handle to some storage space. If the handle is `NIL`, the Sound Input Manager allocates a handle of the largest amount of space that it can find in your application's heap and returns this handle in the `sndHandle` parameter. The Sound Input Manager resizes the handle when the user clicks the Save button in the sound recording dialog box. If the `sndHandle` parameter passed to `SndRecord` is not `NIL`, the Sound Input Manager simply stores the recorded data in the location specified by that handle.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndRecord` function are

Trap macro	Selector
_SoundDispatch	$08040014

RESULT CODES

noErr	0	No error
userCanceledErr	–128	User canceled the operation
siBadSoundInDevice	–221	Invalid sound input device
siUnknownQuality	–232	Unknown quality

SEE ALSO

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of event filter functions.

SndRecordToFile

You can use `SndRecordToFile` to record sound data into a file.

```
FUNCTION SndRecordToFile (filterProc: ProcPtr; corner: Point;
                          quality: OSType;
                          fRefNum: Integer): OSErr;
```

filterProc
  A pointer to a function that determines how user actions in the sound recording dialog box are filtered.

corner   The horizontal and vertical coordinates of the upper-left corner of the sound recording dialog box (in global coordinates).

quality The desired quality of the recorded sound, as described on page 239.

fRefNum The file reference number of an open file to save the audio data in.

DESCRIPTION

The `SndRecordToFile` function works just like `SndRecord` except that it stores the sound input data into a file. The resulting file is in either AIFF or AIFF-C format and contains the information necessary to play the file by using the Sound Manager's `SndStartFilePlay` function. The `SndRecordToFile` function is always called synchronously.

Your application must open the file specified in the `fRefNum` parameter before calling the `SndRecordToFile` function. Your application must close the file sometime after calling `SndRecordToFile`.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SndRecordToFile` function are

Trap macro	**Selector**
`_SoundDispatch`	$07080014

RESULT CODES

`noErr`	0	No error
`userCanceledErr`	–128	User canceled the operation
`siBadSoundInDevice`	–221	Invalid sound input device
`siUnknownQuality`	–232	Unknown quality

Opening and Closing Sound Input Devices

You can use the `SPBOpenDevice` function to open the default sound input device that the user has selected in the Sound In control panel or to open a specific sound input device. You must open a device before you can record from it by using `SPBRecord`, but the Sound Input Manager's high-level routines automatically open the default sound input device. You can close a sound input device by calling the `SPBCloseDevice` function.

SPBOpenDevice

You can use the `SPBOpenDevice` function to open a sound input device.

```
FUNCTION SPBOpenDevice (deviceName: Str255; permission: Integer;
                        VAR inRefNum: LongInt): OSErr;
```

`deviceName`
> The name of the sound input device to open, or the empty string if the default sound input device is to be opened.

`permission`
> A flag that indicates whether subsequent operations with that device are to be read/write or read-only.

`inRefNum`   On exit, if the function is successful, a device reference number for the open sound input device.

DESCRIPTION

The `SPBOpenDevice` function attempts to open a sound input device having the name indicated by the `deviceName` parameter. If `SPBOpenDevice` succeeds, it returns a device reference number in the `inRefNum` parameter. The `permission` parameter indicates whether subsequent operations with that device are to be read/write or read-only. If the device is not already in use, read/write permission is granted; otherwise,

only read-only operations are allowed. To make any recording requests or to call the `SPBSetDeviceInfo` function, read/write permission must be available. Use these constants to request the appropriate permission:

```
CONST
    siReadPermission      = 0;      {open device for reading}
    siWritePermission     = 1;      {open device for reading/writing}
```

You can request that the current default sound input device be opened by passing either a zero-length string or a `NIL` string as the `deviceName` parameter. If only one sound input device is installed, that device is used. Generally you should open the default device unless you specifically want to use some other device. You can get a list of the available devices by calling the `SPBGetIndexedDevice` function.

SPECIAL CONSIDERATIONS

Because the `SPBOpenDevice` function allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBOpenDevice` function are

Trap macro	Selector
_SoundDispatch	$05180014

RESULT CODES

noErr	0	No error
permErr	–54	Device already open for writing
siBadDeviceName	–228	Invalid device name

SPBCloseDevice

You can use the `SPBCloseDevice` function to close a sound input device.

```
FUNCTION SPBCloseDevice (inRefNum: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device to close.

DESCRIPTION

The `SPBCloseDevice` function closes a device that was previously opened by `SPBOpenDevice` and whose device reference number is specified in the `inRefNum` parameter.

SPECIAL CONSIDERATIONS

Because the SPBCloseDevice function moves or purges memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBCloseDevice function are

Trap macro	Selector
_SoundDispatch	$021C0014

RESULT CODES

noErr	0	No error
siBadRefNum	–229	Invalid reference number

Recording Sounds Directly From Sound Input Devices

The Sound Input Manager provides a number of routines that allow you to begin, pause, resume, and stop recording directly from a sound input device. These low-level routines do not display the sound recording dialog box to the user.

SPBRecord

You can use the SPBRecord function to record audio data into memory, either synchronously or asynchronously.

```
FUNCTION SPBRecord (inParamPtr: SPBPtr; asynchFlag: Boolean):
                    OSErr;
```

inParamPtr
    A pointer to a sound input parameter block.

asynchFlag
    A Boolean value that specifies whether the recording occurs asynchronously (TRUE) or synchronously (FALSE).

You specify values and receive return values in the sound input parameter block.

Parameter block

→	inRefNum	LongInt	A reference number of a sound input device.
↔	count	LongInt	The number of bytes of recording.
↔	milliseconds	LongInt	The number of milliseconds of recording.
→	bufferLength	LongInt	The length of the buffer beginning at bufferPtr.

→	bufferPtr	Ptr	A pointer to a buffer for sampled-sound data.
→	completionRoutine	ProcPtr	A pointer to a completion routine.
→	interruptRoutine	ProcPtr	A pointer to an interrupt routine.
→	userLong	LongInt	Free for application's use.
←	error	OSErr	The error value returned after recording.
→	unused1	LongInt	Reserved.

Field descriptions

inRefNum
: The device reference number of the sound input device, as obtained from the SPBOpenDevice function.

count
: On input, the number of bytes to record. If this field indicates a longer recording time than the milliseconds field, then the milliseconds field is ignored. On output, this field indicates the number of bytes actually recorded.

milliseconds
: On input, the number of milliseconds to record. If this field indicates a longer recording time than the count field, then the count field is ignored. On output, this field indicates the number of milliseconds actually recorded.

bufferLength
: The number of bytes in the buffer specified by the bufferPtr parameter. If this buffer length is too small to contain the amount of sampled-sound data specified in the count and milliseconds fields, then recording time is truncated so that the sampled-sound data fits in the buffer.

bufferPtr
: A pointer to the buffer for the sampled-sound data, or NIL if you wish to record sampled-sound data without saving it. On exit, this buffer contains the sampled-sound data, which is interleaved for stereo sound on a sample basis (or on a packet basis if the data is compressed). This buffer contains only sampled-sound data, so if you need a sampled sound header, you should set that up in a buffer before calling SPBRecord and then record into the buffer following the sound header.

completionRoutine
: A pointer to a completion routine. This routine is called when the recording terminates (either after you call the SPBStopRecording function or when the prescribed limit is reached). The completion routine is called only for asynchronous recording.

interruptRoutine
: A pointer to an interrupt routine. The interrupt routine specified in the interruptRoutine field is called by asynchronous recording devices when their internal buffers are full.

userLong
: A long integer that your application can use to pass data to your application's completion or interrupt routines.

error
: On exit, a value greater than 0 while recording unless an error occurs, in which case it contains a value less than 0 that indicates an operating system error. Your application can poll this field to check on the status of an asynchronous recording. If recording terminates without an error, this field contains 0.

unused1   Reserved. You should set this field to 0 before calling `SPBRecord`.

DESCRIPTION

The `SPBRecord` function starts recording into memory from a device specified in a sound input parameter block. The sound data recorded is stored in the buffer specified by the `bufferPtr` and `bufferLength` fields of the parameter block. Recording lasts the longer of the times specified by the `count` and `milliseconds` fields of the parameter block, or until the buffer is filled. Recording is asynchronous if the `asynchFlag` parameter is `TRUE` and the specified sound input device supports asynchronous recording.

If the `bufferPtr` field of the parameter block contains `NIL`, then the `count`, `milliseconds`, and `bufferLength` fields are ignored, and the recording continues indefinitely until you call the `SPBStopRecording` function. In this case, the audio data is not saved anywhere; this feature is useful only if you want to do something in your interrupt routine and do not want to save the audio data. However, if the recording is synchronous and `bufferPtr` is `NIL`, `SPBRecord` returns the result code `siNoBufferSpecified`.

The `SPBRecord` function returns the value that the `error` field of the parameter block contains when recording finishes.

SPECIAL CONSIDERATIONS

You can call the `SPBRecord` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBRecord` function are

Trap macro	Selector
`_SoundDispatch`	$03200014

RESULT CODES

noErr	0	No error
siNoSoundInHardware	–220	No sound input hardware available
siBadSoundInDevice	–221	Invalid sound input device
siNoBufferSpecified	–222	No buffer specified
siDeviceBusyErr	–227	Sound input device is busy

SEE ALSO

For an example of the use of the `SPBRecord` function, see Listing 3-1.

SPBRecordToFile

You can use the `SPBRecordToFile` function to record audio data into a file, either synchronously or asynchronously.

```
FUNCTION SPBRecordToFile (fRefNum: Integer; inParamPtr: SPBPtr;
                          asynchFlag: Boolean): OSErr;
```

fRefNum   The file reference number of an open file in which to place the recorded sound data.

inParamPtr
              A pointer to a sound input parameter block.

asynchFlag
              A Boolean value that specifies whether the recording occurs asynchronously (TRUE) or synchronously (FALSE).

Parameter block

→	inRefNum	LongInt	A reference number of a sound input device.
↔	count	LongInt	The number of bytes of recording.
↔	milliseconds	LongInt	The number of milliseconds of recording.
→	completionRoutine	ProcPtr	A pointer to a completion routine.
→	interruptRoutine	ProcPtr	Unused.
→	userLong	LongInt	Free for application's use.
←	error	OSErr	The error value returned after recording.
→	unused1	LongInt	Reserved.

Field descriptions

inRefNum   The device reference number of the sound input device, as obtained from the `SPBOpenDevice` function.

count   On input, the number of bytes to record. If this field indicates a longer recording time than the `milliseconds` field, then the `milliseconds` field is ignored. On output, the number of bytes actually recorded.

milliseconds   On input, the number of milliseconds to record. If this field indicates a longer recording time than the `count` field, then the `count` field is ignored. On output, the number of milliseconds actually recorded.

completionRoutine
               A pointer to a completion routine. This routine is called when the recording terminates (after you call the `SPBStopRecording` function, when the prescribed limit is reached, or after an error occurs). The completion routine is called only for asynchronous recording.

interruptRoutine
               Unused. You should set this field to `NIL` before calling `SPBRecordToFile`.

userLong   A long integer that your application can use to pass data to your
                  application's completion or interrupt routines.

error   On exit, the error that occurred during recording. This field contains
                  the number 1 while recording unless an error occurs, in which case it
                  contains a value less than 0 that indicates an operating system error.
                  Your application can poll this field to check on the status of an
                  asynchronous recording. If recording terminates without an error,
                  this field contains 0.

unused1   Reserved. You should set this field to 0 before calling the
                  SPBRecordToFile function.

DESCRIPTION

The SPBRecordToFile function starts recording from the specified device into a file.
The sound data recorded is simply stored in the file, so it is up to your application to
insert whatever headers are needed to play the sound with the Sound Manager. Your
application must open the file specified by the fRefNum parameter with write access
before calling SPBRecordToFile, and it must eventually close that file.

The fields in the parameter block specified by the inParamPtr parameter are identical to
the fields in the parameter block passed to the SPBRecord function, except that the
bufferLength and bufferPtr fields are not used. The interruptRoutine field is
ignored by SPBRecordToFile because SPBRecordToFile copies data returned by the
sound input device driver to disk during the sound input interrupt routine, but you
should initialize this field to NIL.

The SPBRecordToFile function writes samples to disk in the same format that they are
read in from the sound input device. If compression is enabled, then the samples written
to the file are compressed. Multiple channels of sound are interleaved on a sample basis
(or, for compressed sound data, on a packet basis). When you are recording 8-bit audio
data to an AIFF file, you must set the siTwosComplementOnOff flag to so that the data
is stored on disk in the two's-complement format. If you don't store the data in this
format, it sounds distorted when you play it back.

If any errors occur during the file writing process, recording is suspended. All File
Manager errors are returned through the function's return value if the routine is called
synchronously. If the routine is called asynchronously and the completion routine is not
NIL, the completion routine is called and is passed a single parameter on the stack that
points to the sound input parameter block; any errors are returned in the error field of
the sound input parameter block.

The SPBRecordToFile function returns the value that the error field of the parameter
block contains when recording finishes.

SPECIAL CONSIDERATIONS

Because the SPBRecordToFile function moves or purges memory, you should not call
it at interrupt time.

Sound Input Manager

3

The trap macro and routine selector for the SPBRecordToFile function are

Trap macro	Selector
_SoundDispatch	$04240014

RESULT CODES

noErr	0	No error
permErr	–54	Attempt to open locked file for writing
siNoSoundInHardware	–220	No sound input hardware available
siBadSoundInDevice	–221	Invalid sound input device
siHardDriveTooSlow	–224	Hard drive too slow to record

SPBPauseRecording

You can use the SPBPauseRecording function to pause recording from a sound input device.

```
FUNCTION SPBPauseRecording (inRefNum: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the SPBOpenDevice function.

DESCRIPTION

The SPBPauseRecording function pauses recording from the device specified by the inRefNum parameter. The recording must be asynchronous for this call to have any effect.

SPECIAL CONSIDERATIONS

You can call the SPBPauseRecording function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBPauseRecording function are

Trap macro	Selector
_SoundDispatch	$02280014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device

SPBResumeRecording

You can use the `SPBResumeRecording` function to resume recording from a sound input device.

```
FUNCTION SPBResumeRecording (inRefNum: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the `SPBOpenDevice` function.

DESCRIPTION

The `SPBResumeRecording` function resumes recording from the device specified by the `inRefNum` parameter. Recording on that device must previously have been paused by a call to the `SPBPauseRecording` function for `SPBResumeRecording` to have any effect.

SPECIAL CONSIDERATIONS

You can call the `SPBResumeRecording` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBResumeRecording` function are

Trap macro	Selector
_SoundDispatch	$022C0014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	−221	Invalid sound input device

SPBStopRecording

You can use the `SPBStopRecording` function to end a recording from a sound input device.

```
FUNCTION SPBStopRecording (inRefNum: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the `SPBOpenDevice` function.

DESCRIPTION

The `SPBStopRecording` function stops recording from the device specified by the `inRefNum` parameter. The recording must be asynchronous for `SPBStopRecording`

to have any effect. When you call SPBStopRecording, the sound input completion routine specified in the completionRoutine field of the sound input parameter block is called and the error field of that parameter block is set to abortErr. If you are writing a device driver, you will receive a KillIO Status call. See the section "Writing a Sound Input Device Driver" beginning on page 223 for more information.

SPECIAL CONSIDERATIONS

You can call the SPBStopRecording function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBStopRecording function are

Trap macro	Selector
_SoundDispatch	$02300014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device

SPBGetRecordingStatus

You can use SPBGetRecordingStatus to obtain recording status information about a sound input device.

```
FUNCTION SPBGetRecordingStatus (inRefNum: LongInt;
                                VAR recordingStatus: Integer;
                                VAR meterLevel: Integer;
                                VAR totalSamplesToRecord: LongInt;
                                VAR numberOfSamplesRecorded: LongInt;
                                VAR totalMsecsToRecord: LongInt;
                                VAR numberOfMsecsRecorded: LongInt):
                                OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the SPBOpenDevice function.

recordingStatus
            The status of the recording. While the input device is recording, this parameter is set to a number greater than 0. When a recording terminates without an error, this parameter is set to 0. When an error occurs during recording or the recording has been terminated by a call to the SPBStopRecording function, this parameter is less than 0 and contains an error code.

meterLevel

The current input signal level. This level ranges from 0 to 255.

totalSamplesToRecord

The total number of samples to record, including those samples already recorded.

numberOfSamplesRecorded

The number of samples already recorded.

totalMsecsToRecord

The total duration of recording time, including recording time already elapsed.

numberOfMsecsRecorded

The amount of recording time that has elapsed.

DESCRIPTION

The SPBGetRecordingStatus function returns, in its second through seventh parameters, information about the recording on the device specified by the inRefNum parameter.

SPECIAL CONSIDERATIONS

You can call the SPBGetRecordingStatus function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBGetRecordingStatus function are

Trap macro	Selector
_SoundDispatch	$0E340014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device

Manipulating Device Settings

You can use the two functions SPBGetDeviceInfo and SPBSetDeviceInfo to read and change the settings of a sound input device.

SPBGetDeviceInfo

You can use the SPBGetDeviceInfo function to get information about the settings of a sound input device.

```
FUNCTION SPBGetDeviceInfo (inRefNum: LongInt; infoType: OSType;
                                 infoData: Ptr): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the SPBOpenDevice function.

infoType   A sound input device information selector that specifies the type of information you need.

infoData   A pointer to a buffer in which information should be returned. This buffer must be large enough for the type of information specified in the infoType parameter.

DESCRIPTION

The SPBGetDeviceInfo function returns information about the sound input device specified by the inRefNum parameter. The type of information you want is specified in the infoType parameter. The available sound input device information selectors are listed in "Sound Input Device Information Selectors" beginning on page 229. The information is copied into the buffer specified by the infoData parameter.

SPECIAL CONSIDERATIONS

Because the SPBGetDeviceInfo function might move memory, you should not call it at interrupt time. Check the selector description of the selector you want to use to see if it moves memory before calling the SPBGetDeviceInfo function. Most of the selectors do not move memory and are therefore safe to use at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBGetDeviceInfo function are

Trap macro	Selector
_SoundDispatch	$06380014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device
siUnknownInfoType	–231	Unknown type of information

SEE ALSO

Listing 3-2 on page 222 shows an example that uses the SPBGetDeviceInfo function to get the name of a sound input device driver.

SPBSetDeviceInfo

You can use the SPBSetDeviceInfo function to set information in a sound input device.

```
FUNCTION SPBSetDeviceInfo (inRefNum: LongInt; infoType: OSType;
                           infoData: Ptr): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the SPBOpenDevice function.

infoType   A sound input device information selector that specifies the type of information you need.

infoData   A pointer to a buffer. This buffer can contain information on entry, and information might be returned on exit. This buffer must be large enough for the type of information specified in the infoType parameter, and the data in the buffer must be set to appropriate values if information needs to be passed in to the SPBSetDeviceInfo function.

DESCRIPTION

The SPBSetDeviceInfo function sets information about the sound input device specified by the inRefNum parameter, based on the data in the buffer specified by the infoData parameter.

The type of setting you wish to change is specified in the infoType parameter. The sound input device information selectors are listed in "Sound Input Device Information Selectors" beginning on page 229.

SPECIAL CONSIDERATIONS

Because the SPBSetDeviceInfo function might move memory, you should not call it at interrupt time. Check the selector description of the selector you want to use to see if it moves memory before calling the SPBGetDeviceInfo function. Most of the selectors do not move memory and are therefore safe to use at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBSetDeviceInfo function are

Trap macro   **Selector**
_SoundDispatch $063C0014

RESULT CODES

noErr	0	No error
permErr	–54	Attempt to open locked file for writing
siBadSoundInDevice	–221	Invalid sound input device
siDeviceBusyErr	–227	Sound input device is busy
siUnknownInfoType	–231	Unknown type of information

Constructing Sound Resource and File Headers

The Sound Input Manager provides two functions, SetupSndHeader and SetupAIFFHeader, to help you set up headers for sound resources and sound files.

SetupSndHeader

You can use the SetupSndHeader function to construct a sound resource containing sampled sound that can be passed to the SndPlay function.

```
FUNCTION SetupSndHeader (sndHandle: Handle;
                         numChannels: Integer;
                         sampleRate: Fixed;
                         sampleSize: Integer;
                         compressionType: OSType;
                         baseFrequency: Integer;
                         numBytes: LongInt;
                         VAR headerLen: Integer): OSErr;
```

sndHandle   A handle to a block of memory that is at least large enough to store the sound resource header information. The handle is not resized in any way upon successful completion of SetupSndHeader. The SetupSndHeader function simply fills the relocatable block specified by this parameter with the header information needed for a format 1 'snd ' resource, including the sound resource header, the list of sound commands, and a sampled sound header. It is your application's responsibility to append the desired sampled-sound data.

numChannels
            The number of channels for the sound; one channel is equivalent to monaural sound and two channels are equivalent to stereo sound.

sampleRate
            The rate at which the sound was recorded. The sample rate is declared as a Fixed data type. In order to accommodate sample rates greater than 32 kHz, the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.

sampleSize
            The sample size for the original sound (that is, bits per sample).

compressionType

> The compression type for the sound ('NONE', 'MAC3', 'MAC6', or other third-party types).

baseFrequency

> The base frequency for the sound, expressed as a MIDI note value.

numBytes   The number of bytes of audio data that are to be stored in the handle. (This value is not necessarily the same as the number of samples in the sound.)

headerLen   On exit, the size (in bytes) of the 'snd ' resource header that is created. In no case will this length exceed 100 bytes. This field allows you to put the audio data right after the header in the relocatable block specified by the sndHandle parameter. The value returned depends on the type of sound header created.

DESCRIPTION

The SetupSndHeader function creates a format 1 'snd ' resource for a sampled sound. The resource contains a sound resource header that links the sound to the sampled synthesizer, a single sound command (a bufferCmd command to play the accompanying data), and a sampled sound header. You can use SetupSndHeader to construct a sampled sound header that can be passed to the Sound Manager's SndPlay function or stored as an 'snd ' resource. After calling the SetupSndHeader function, your application should place the sampled-sound data directly after the sampled sound header so that, in essence, the sampled sound header's final field contains the sound data.

The sampled sound is in one of three formats depending on several of the parameters passed. Table 3-1 shows how SetupSndHeader determines what kind of sound header to create.

Table 3-1   The sampled sound header format used by SetupSndHeader

compressionType	numChannels	sampleSize	Sampled sound header format
'NONE'	1	8	SoundHeader
'NONE'	1	16	ExtSoundHeader
'NONE'	2	any	ExtSoundHeader
not 'NONE'	any	any	CmpSoundHeader

A good way to use this function is to create a handle in which you want to store a sampled sound, then call SetupSndHeader with the numBytes parameter set to 0 to see how much room the header for that sound will occupy and hence where to append the audio data. Then record the data into the handle and call SetupSndHeader again with numBytes set to the correct amount of sound data recorded. The handle filled out in this way can be passed to SndPlay to play the sound.

SPECIAL CONSIDERATIONS

You cannot call the `SetupSndHeader` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetupSndHeader` function are

Trap macro	Selector
`_SoundDispatch`	$0D480014

RESULT CODES

`noErr`	0	No error
`siInvalidCompression`	–223	Invalid compression type

SEE ALSO

For an example that uses the `SetupSndHeader` function to set up a sound header before recording, see Listing 3-1 on page 217.

SetupAIFFHeader

You can use the `SetupAIFFHeader` function to set up a file that can subsequently be played by `SndStartFilePlay`.

```
FUNCTION SetupAIFFHeader (fRefNum: Integer;
                          numChannels: Integer;
                          sampleRate: Fixed;
                          sampleSize: Integer;
                          compressionType: OSType;
                          numBytes: LongInt;
                          numFrames: LongInt): OSErr;
```

fRefNum   A file reference number of a file that is open for writing.

numChannels
              The number of channels for the sound; one channel is equivalent to monaural sound and two channels are equivalent to stereo sound.

sampleRate
              The rate at which the sound was recorded. The sample rate is declared as a `Fixed` data type. In order to accommodate sample rates greater than 32 kHz, the most significant bit is not treated as a sign bit; instead, that bit is interpreted as having the value 32,768.

sampleSize
              The sample size for the original sound (that is, bits per sample).

compressionType
: The compression type for the sound ('NONE', 'MAC3', 'MAC6', or other third-party types).

numBytes
: The number of bytes of audio data that are to be stored in the Common Chunk of the AIFF or AIFF-C file.

numFrames
: The number of sample frames for the sample sound. If you are using a compression type defined by Apple, you can pass 0 in this field and the appropriate value for this field will be computed automatically.

DESCRIPTION

The SetupAIFFHeader function creates an AIFF or AIFF-C file header, depending on the parameters passed to it:

■ Uncompressed sounds of any type are stored in AIFF format (that is, the compressionType parameter is 'NONE').

■ Compressed sounds of any type are stored in AIFF-C format (that is, the compressionType parameter is different from 'NONE').

Note
The SetupAIFFHeader function might format a sound file as an AIFF file even if the File Manager file type of a file is 'AIFC'. The Sound Manager will still play such files correctly. ◆

The AIFF header information is written starting at the current file position of the file specified by the fRefNum parameter, and the file position is left at the end of the header upon completion. The SetupAIFFHeader function creates a Form Chunk, a Format Version Chunk, a Common Chunk, and a Sound Data chunk, but it does not put any sound data at the end of the Sound Data Chunk.

A good way to use this routine is to create a file that you want to store a sound in, then call SetupAIFFHeader with numBytes set to 0 to position the file to be ready to write the audio data. Then record the data to the file, set the file position to the beginning of the file, and call SetupAIFFHeader again with numBytes set to the correct amount of sound data recorded. The file created in this way can be passed to the SndStartFilePlay function to play the sound.

SPECIAL CONSIDERATIONS

If recording produces an odd number of bytes of sound data, you must add a pad byte to make the total number of bytes even.

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetupAIFFHeader` function are

Trap macro	Selector
`_SoundDispatch`	$0B4C0014

RESULT CODES

`noErr`	0	No error
`siInvalidCompression`	–223	Invalid compression type

Registering Sound Input Devices

Sound input device drivers must call the `SPBSignInDevice` function to register with the Sound Input Manager before they can use its sound input services. You might call this routine at system startup time from within an extension to install a sound input device driver. Your application can generate a list of registered sound input devices by using the `SPBGetIndexedDevice` function. You can cancel the registration of your driver, thus removing it from the Sound control panel and making it inaccessible, by calling the `SPBSignOutDevice` function.

SPBSignInDevice

You can register a sound input device by calling the `SPBSignInDevice` function.

```
FUNCTION SPBSignInDevice (deviceRefNum: Integer;
                              deviceName: Str255): OSErr;
```

`deviceRefNum`
                    The device driver reference number of the sound input device to register with the Sound Input Manager.

`deviceName`
                    The device's name as it is to appear to the user in the Sound In control panel (which is not the name of the driver used by the Device Manager).

DESCRIPTION

The `SPBSignInDevice` function registers with the Sound Input Manager the device whose driver reference number is `deviceRefNum`.

The `deviceName` parameter specifies this device's name as it is to appear to the user in the Sound In control panel (which is not the name of the driver itself). Accordingly, the name should be as descriptive as possible. You should call `SPBSignInDevice` after you have already opened your driver by calling normal Device Manager routines.

SPECIAL CONSIDERATIONS

Because the SPBSignInDevice function moves or purges memory, you should not call it at interrupt time. You can, however, call it at system startup time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the SPBSignInDevice function are

Trap macro	Selector
_SoundDispatch	$030C0014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	−221	Invalid sound input device

SPBGetIndexedDevice

You can use the SPBGetIndexedDevice function to help generate a list of sound input devices.

```
FUNCTION SPBGetIndexedDevice (count: Integer;
                              VAR deviceName: Str255;
                              VAR deviceIconHandle: Handle):
                              OSErr;
```

count   The index number of the sound input device you wish to obtain information about.

deviceName
                On exit, the name of the sound input device specified by the count parameter.

deviceIconHandle
                On exit, a handle to the icon of the sound input device specified by the count parameter. The memory for this icon is allocated automatically, but your application must dispose of it.

DESCRIPTION

The SPBGetIndexedDevice function returns the name and icon of the device whose index is specified in the count parameter. Your application can create a list of sound input devices by calling this function with a count starting at 1 and incrementing it by 1 until the function returns siBadSoundInDevice.

Because the Sound In control panel allows the user to select a sound input device, most applications should not use this function. Your application might need to use this function if it allows the user to record from more than one sound input device at once.

SPECIAL CONSIDERATIONS

Because the `SPBGetIndexedDevice` function allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBGetIndexedDevice` function are

Trap macro	Selector
`_SoundDispatch`	`$05140014`

RESULT CODES

`noErr`	0	No error
`siBadSoundInDevice`	–221	Invalid sound input device

SPBSignOutDevice

You can use the `SPBSignOutDevice` function to cancel the registration of a device you have previously registered with the `SPBSignInDevice` function.

```
FUNCTION SPBSignOutDevice (deviceRefNum: Integer): OSErr;
```

`deviceRefNum`
    The driver reference number of the device you wish to sign out.

DESCRIPTION

The `SPBSignOutDevice` function cancels the registration of the device whose driver reference number is `deviceRefNum`; the device is unregistered from the Sound Input Manager's list of available sound input devices and no longer appears in the Sound In control panel.

Ordinarily, you should not need to use the `SPBSignOutDevice` function. You might use it if your device driver detects that a sound input device is not functioning correctly or has been disconnected.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBSignOutDevice` function are

Trap macro	Selector
`_SoundDispatch`	`$01100014`

RESULT CODES

`noErr`	0	No error
`siBadSoundInDevice`	−221	Invalid sound input device
`siDeviceBusyErr`	−227	Sound input device is busy

Converting Between Milliseconds and Bytes

The Sound Input Manager provides two routines that allow you to convert between millisecond and byte recording values.

SPBMilliSecondsToBytes

You can use the `SPBMilliSecondsToBytes` function to determine how many bytes a recording of a certain duration will use.

```
FUNCTION SPBMilliSecondsToBytes (inRefNum: LongInt;
                                 VAR milliseconds: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the `SPBOpenDevice` function.

milliseconds
             On entry, the duration of the recording in milliseconds. On exit, the number of bytes that sampled-sound data would occupy for a recording of the specified duration on the device specified by the `inRefNum` parameter.

DESCRIPTION

The `SPBMilliSecondsToBytes` function reports how many bytes are required to store a recording of duration `milliseconds`, given the input device's current sample rate, sample size, number of channels, and compression factor.

SPECIAL CONSIDERATIONS

You can call the `SPBMilliSecondsToBytes` function at interrupt time.

The trap macro and routine selector for the `SPBMilliSecondsToBytes` function are

Trap macro	Selector
_SoundDispatch	$04400014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device

SPBBytesToMilliSeconds

You can use the `SPBBytesToMilliSeconds` function to determine the maximum duration of a recording that can fit in a buffer of a certain size.

```
FUNCTION SPBBytesToMilliSeconds (inRefNum: LongInt;
                                    VAR byteCount: LongInt): OSErr;
```

inRefNum   The device reference number of the sound input device, as obtained from the `SPBOpenDevice` function.

byteCount   On entry, a value in bytes. On exit, the number of milliseconds of recording on the device specified by the `inRefNum` parameter that would be necessary to fill a buffer of such a size.

DESCRIPTION

The `SPBBytesToMilliSeconds` function reports how many milliseconds of audio data can be recorded in a buffer that is `byteCount` bytes long, given the input device's current sample rate, sample size, number of channels, and compression factor.

SPECIAL CONSIDERATIONS

You can call the `SPBBytesToMilliSeconds` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBBytesToMilliSeconds` function are

Trap macro	Selector
_SoundDispatch	$04440014

RESULT CODES

noErr	0	No error
siBadSoundInDevice	–221	Invalid sound input device

Obtaining Information

The `SPBVersion` function allows you to determine the version of the Sound
Input Manager.

SPBVersion

You can use the `SPBVersion` function to determine the version of the sound input tools
available on a machine.

```
FUNCTION SPBVersion: NumVersion;
```

DESCRIPTION

The `SPBVersion` function returns a version number that contains the same information
as in the first 4 bytes of a `'vers'` resource or a `NumVersion` data type. For a description
of the version record, see the chapter "Sound Manager" in this book.

SPECIAL CONSIDERATIONS

You can call the `SPBVersion` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SPBVersion` function are

Trap macro	Selector
`_SoundDispatch`	$00000014

SEE ALSO

For a complete discussion of `'vers'` resources, see the chapter "Finder Interface" in
Inside Macintosh: Macintosh Toolbox Essentials.

Application-Defined Routines

This section describes the routines that your application or device driver might need to
define. Your application can define a sound input completion routine to perform an
action when recording finishes, and your application can define a sound input interrupt
routine to manipulate sound data during recording.

Sound Input Completion Routines

You can specify a sound input completion routine in the `completionRoutine` field of a sound input parameter block that your application uses to initiate asynchronous recording directly from a device.

MySICompletionRoutine

A sound input completion routine has the following syntax:

```
PROCEDURE MySICompletionRoutine (inParamPtr: SPBPtr);
```

inParamPtr
> A pointer to the sound input parameter block that was used to initiate an asynchronous recording.

DESCRIPTION

The Sound Input Manager executes your sound input completion routine after recording terminates either because your application has called the `SPBStopRecording` function or because the prescribed limit is reached. The completion routine is called only for asynchronous recording.

A common use of a sound input completion routine is to set a global variable that alerts the application that it should dispose of a sound input parameter block that it had allocated for an asynchronous sound recording.

SPECIAL CONSIDERATIONS

Because a sound input completion routine is executed at interrupt time, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

If your sound input completion routine accesses your application's global variables, it must ensure that the A5 register contains the address of the boundary between the application global variables and the application parameters. Your application can pass the value of the A5 register to the sound input completion routine in the `userLong` field of the sound input parameter block. For more information on ensuring the validity of the A5 register, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.*

Your sound input completion routine can determine whether an error occurred during recording by examining the `error` field of the sound input parameter block specified by `inParamPtr`. Your sound input completion routine can change the value of that field to alert the application that some other error has occurred.

ASSEMBLY-LANGUAGE INFORMATION

Because a sound input completion routine is called at interrupt time, it must preserve all registers other than A0–A1 and D0–D2.

RESULT CODES

noErr	0	No error
abortErr	–27	Asynchronous recording was cancelled
siNoSoundInHardware	–220	No sound input hardware available
siBadSoundInDevice	–221	Invalid sound input device
siNoBufferSpecified	–222	No buffer specified
siDeviceBusyErr	–227	Sound input device is busy

Sound Input Interrupt Routines

You can specify a sound input interrupt routine in the `interruptRoutine` field of the sound input parameter block that your application uses to initiate asynchronous recording directly from a device. Because the `SPBRecordToFile` function uses sound input interrupt routines to enable it to record sound data to disk during recording, you can use sound input interrupt routines only with the `SPBRecord` function.

MySIInterruptRoutine

A sound input interrupt routine has the following syntax:

```
PROCEDURE MySIInterruptRoutine;
```

DESCRIPTION

A sound input device driver executes the sound input interrupt routine associated with an asynchronous sound recording whenever the driver's internal buffers are full. The internal buffers contain raw samples taken directly from the input device. The interrupt routine can thus modify the samples in the buffer in any way it requires. After your sound input interrupt routine finishes processing the data, the sound input device driver compresses the data (if compression is enabled) and copies the data into your application's buffer.

SPECIAL CONSIDERATIONS

If your sound input interrupt routine accesses your application's global variables, it must ensure that the A5 register contains the address of the boundary between the application global variables and the application parameters. Your application can pass the value of the A5 register to the sound input interrupt routine in the `userLong` field of the sound input parameter block. For more information on ensuring the validity of the A5 register, see the chapter "Memory Management Utilities" in *Inside Macintosh: Memory.*

ASSEMBLY-LANGUAGE INFORMATION

Sound input interrupt routines are sometimes written in assembly language to maximize real-time performance in recording sound. On entry, registers are set up as follows:

Registers on entry

A0	Address of the sound parameter block passed to `SPBRecord`
A1	Address of the start of the sample buffer
D0	Peak amplitude for sample buffer if metering is on
D1	Size of the sample buffer in bytes

If you write a sound input interrupt routine in a high-level language like Pascal or C, you might need to write inline code to copy variables from the registers into local variables that your application defines.

Because a sound input interrupt routine is called at interrupt time, it must preserve all registers.

Sound Components

This chapter describes sound components, which are code modules used by the
Sound Manager to manipulate audio data or to communicate with sound output devices.
Current versions of the Sound Manager allow you to write two kinds of
sound components:

■ compression and decompression components (codecs), which allow you to implement
  audio data compression and decompression algorithms different from those provided
  by the Sound Manager's MACE (Macintosh Audio Compression and Expansion)
  capabilities

■ sound output device components, which send audio data directly to sound output
  devices

You need to read this chapter only if you are developing a sound output device or if you
want to implement a custom compression and decompression scheme for audio data. For
example, you might write a codec to handle 16-bit audio data compression and
decompression. (The MACE algorithms currently compress and expand only 8-bit data at
ratios of 3:1 and 6:1.)

IMPORTANT

Sound components are loaded and managed by the Sound Manager
and operate transparently to applications. Applications that want to
create sounds must use Sound Manager routines to do so. The routines
described in this chapter are intended for use exclusively by sound
components. ▲

To use this chapter, you should already be familiar with the general operation of the
Sound Manager, as described in the chapter "Introduction to Sound on the Macintosh" in
this book. Because sound components are components, you also need to be familiar with
the Component Manager, described in *Inside Macintosh: More Macintosh Toolbox*. If you are
developing a sound output device component, you need to be familiar with the process
of installing a driver and handling interrupts created by your hardware device. See *Inside
Macintosh: Devices* for complete information on devices and device drivers.

If you're developing a sound output device, you might also need to write a control panel
extension that installs a custom subpanel into the Sound control panel. For example, your

subpanel could allow the user to set various characteristics of the sound your output device is creating. For complete information on writing control panel subpanels, see the chapter "Control Panel Extensions" in *Inside Macintosh: Operating System Utilities*.

This chapter begins with a general description of sound components and how they are managed by the Sound Manager. Then it provides instructions on how to write a sound component. The section "Sound Components Reference" beginning on page 286 describes the sound component selectors your component might need to handle and the component-defined routines that your sound component should call in response to those the sound component selectors. It also describes a small number of Sound Manager utility routines that your sound component can use.

Note
This chapter provides all source code examples and reference materials in C. ◆

About Sound Components

A **sound component** is a component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. Sound components provide the foundation for the modular, device-independent sound architecture introduced with Sound Manager version 3.0. This section provides a description of sound components and shows how they are managed by the Sound Manager. For specific information on creating a sound component, see "Writing a Sound Component" beginning on page 273.

Sound Component Chains

Prior to version 3.0, the Sound Manager performed all audio data processing internally, using its own filters to decompress audio data, convert sample rates, mix separate sound channels, and so forth. This effectively rendered it difficult, if not impossible, to add other data modification filters to process the audio data. (The now-obsolete method of installing a sound modifier with the `SndAddModifier` routine did not work reliably.) More importantly, the Sound Manager was responsible for managing the entire stream of audio data, from the application to the available sound-producing audio hardware. This made it very difficult to support new sound output devices.

In versions 3.0 and later, the Sound Manager provides a new audio data processing architecture based on components, illustrated in Figure 4-1. The fundamental idea is that the process of playing a sound can be divided into a number of specific steps, each of which has well-defined inputs and outputs. Figure 4-1 shows the steps involved in playing an 11 kHz compressed sampled sound resource on a Macintosh II computer.

An application sends the compressed sound data to the Sound Manager, which constructs an appropriate **sound component chain** that links the unprocessed audio data to the sound components required to modify the data into a form that can be sent to the current sound output device. As you can see in Figure 4-1, the Sound Manager links

together sound components that, in sequence, expand the compressed sound data into audio samples, convert the sample rate from 11 kHz to 22 kHz, mix those samples with samples from any other sound channels that might be playing, and then write the samples to the available audio hardware (in this case, the FIFO buffer in the Apple Sound Chip).

IMPORTANT

The Sound Manager itself converts both wave-table data and square-wave data into sampled-sound data before sending the data into a chain of sound components. As a result, sound components need to be concerned only with sampled-sound data. ▲

Figure 4-1   The component-based sound architecture
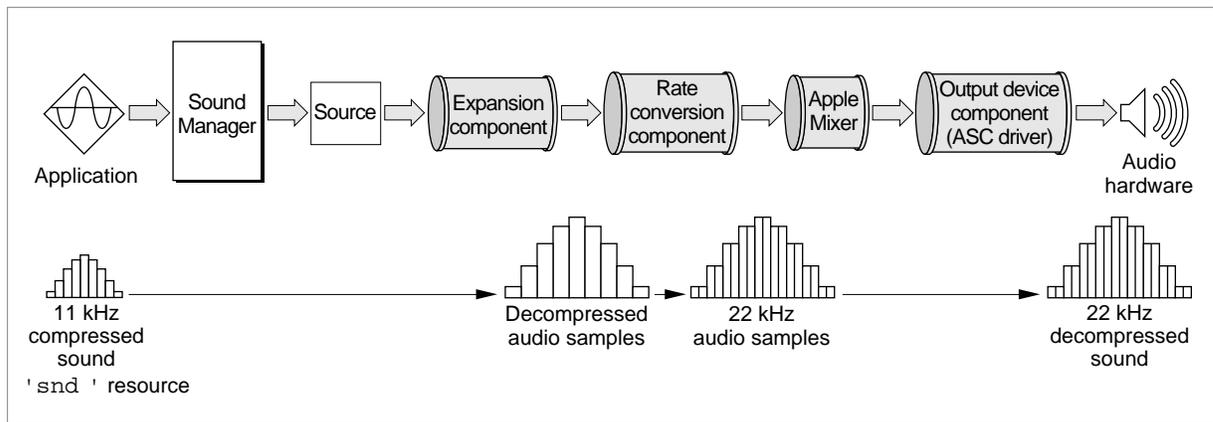
The components in a component chain may vary, depending both on the format of the audio data sent to the Sound Manager by an application and on the capabilities of the current sound output device. The chain shown in Figure 4-1 is necessary to handle the compressed 11 kHz sound because the Apple Sound Chip can handle only 22 kHz noncompressed sampled-sound data. Other sound output devices may be able to do more processing internally, thereby reducing the amount of processing required by the sound component chain. For instance, a DSP-based sound card might be capable of converting sample rates itself. In that case, the Sound Manager would not install the rate conversion component into the sound component chain. The resulting sound component chain is shown in Figure 4-2.

4

Sound Components

Figure 4-2   A component chain for audio hardware that can convert sample rates
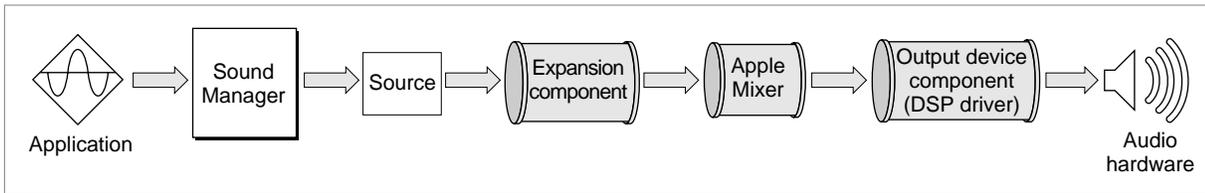
The principal function of a sound component is to transfer data from the source down the chain of sound components while performing some specific modification on the data. It does this by getting a block of data from its **source component** (the component that immediately precedes it in the chain). The sound component then processes that data and stores it in the component's own private buffers. The next component can then get that processed data, perform its own modifications, and pass the data to the next component in the chain. Eventually, the audio data flows through the Apple Mixer (described in the next section) to the **sound output device component,** which sends the data to the current sound output device.

Notice that only the sound output device component communicates directly with the sound output hardware. This insulates all other sound components from having to know anything about the current sound output device. Rather, those components (sometimes called **utility components**) can simply operate on a stream of bytes.

The Sound Manager provides sound output device components for all sound output devices built into Macintosh computers. It also provides utility components for many typical kinds of audio data manipulation, including

■ sample rate conversion

■ audio data expansion

■ sample size conversion

■ format conversion (for example, converting offset binary data to two's complement)

Currently, you can write sound output device components to handle communication with your own sound output devices. You can also write utility components to handle custom compression and expansion schemes. You cannot currently write any other kind of utility component.

The Apple Mixer

As you've seen, most sound components take a single source of audio data and modify it in some way, thereby producing a single output stream of audio data. There is one special sound component, known as the **Apple Mixer component** (or, more briefly, the **Apple Mixer**), that is able to handle more than one input data stream. Its function is precisely to mix together all open channels of sound data into a single output stream, as shown in Figure 4-3.

Figure 4-3   Mixing multiple channels of sound
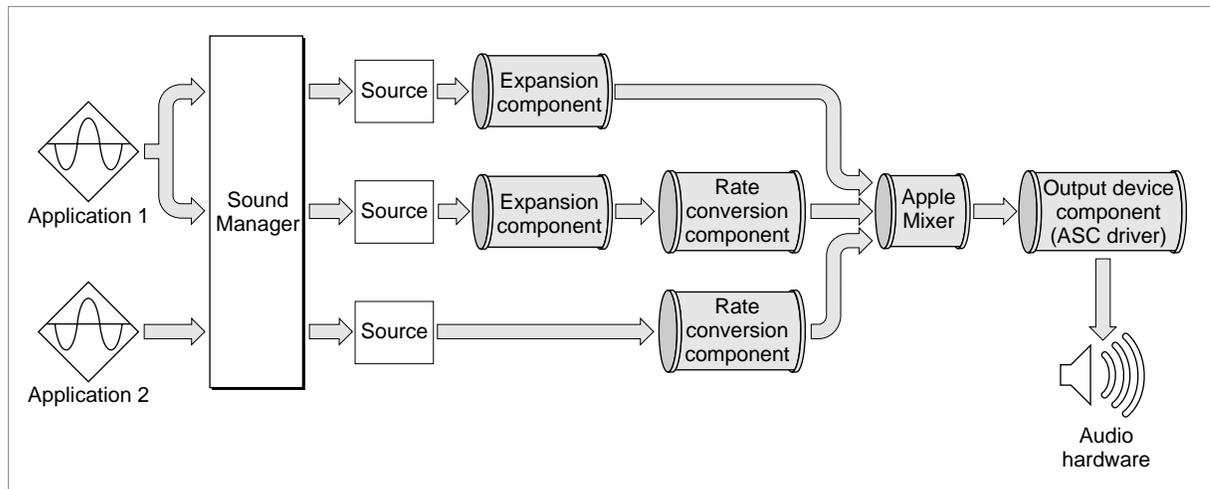
The Apple Mixer has a more general function also, namely to construct the sound component chain required to process audio data from a given sound source into a format that can be handled by a particular sound output device. The Apple Mixer always feeds its output directly to the sound output device component, which sends the data to its associated audio hardware. After creating the component chain, the Apple Mixer assigns it a **source ID,** a 4-byte token that provides a unique reference to the component chain. The Apple Mixer is actually created by the sound output device component, when that component calls the Sound Manager's OpenMixerSoundComponent function.

In addition to creating sound component chains and mixing their data, the Apple Mixer can control the volume and stereo panning of a particular sound channel. Some sound output devices might be able to provide these capabilities as well. Indeed, some sound output devices might even be able to mix the data in multiple sound channels. In those cases, the sound output device component can call the OpenMixerSoundComponent function once for each sound source it wants to manage. The result is a separate instance of the Apple Mixer for each sound source, as shown in Figure 4-4.

Figure 4-4   A sound output device component that can mix sound channels
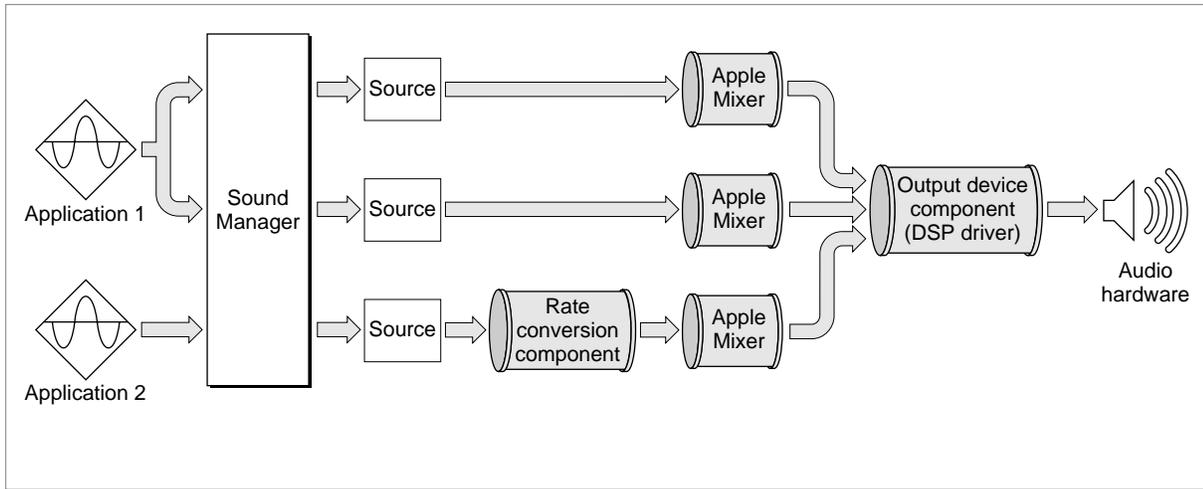
The sound output device component can instruct each instance of the Apple Mixer to pass all the sound data through unprocessed, thereby allowing the output device to perform the necessary processing and mixing. In this case, the Apple Mixer consumes virtually no processing time. The Apple Mixer must, however, still be present to set up the sound component chain and to assign a source ID to each sound source.

The Data Stream

A sound component is a standalone code resource that performs some signal processing function or communicates with a sound output device. All sound components have a standard programming interface and local storage that allows them to be connected together in series to perform a wide range of audio data processing tasks. As previously indicated, all sound components (except for mixer components and some sound output device components) accept a single stream of input data and produce a single stream of output data.

The Sound Manager sends your sound component information about its input stream by passing it the address of a **sound component data record,** defined by the `SoundComponentData` data type.

```
typedef struct {
   long          flags;         /*sound component flags*/
   OSType        format;        /*data format*/
   short         numChannels;   /*number of channels in data*/
   short         sampleSize;    /*size of a sample*/
   UnsignedFixed sampleRate;    /*sample rate*/
   long          sampleCount;   /*number of samples in buffer*/
```

```
  Byte            *buffer;        /*location of data*/
  long            reserved;       /*reserved*/
} SoundComponentData, *SoundComponentDataPtr;
```

The `buffer` field points to the buffer of input data. The other fields define the format of that data. For example, the sample size and rate are passed in the `sampleSize` and `sampleRate` fields, respectively. A utility component should modify the data in that buffer and then write the processed data into an internal buffer. Then it should fill out a sound component data record and pass its address back to the Sound Manager, which will then pass it on to the next sound component in the chain. Eventually, the audio data passes through all utility components in the chain, through the Apple Mixer and the sound output device component, down to the audio hardware.

Writing a Sound Component

A sound component is a component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. Because a sound component is a component, it must be able to respond to standard selectors sent by the Component Manager. In addition, a sound component must handle other selectors specific to sound components. This section describes how to write a sound component.

Creating a Sound Component

A sound component is a component. It contains a number of resources, including icons, strings, and the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a sound component must contain code to handle required selectors passed to it by the Component Manager as well as selectors specific to the sound component.

Note
For complete details on components and their structure, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.* This section provides specific information about sound components. ◆

The component resource binds together all the relevant resources contained in a component; its structure is defined by the `ComponentResource` data type.

```
struct ComponentResource {
   ComponentDescription    cd;
   ResourceSpec            component;
   ResourceSpec            componentName
   ResourceSpec            componentInfo;
   ResourceSpec            componentIcon;
};
```

The `component` field specifies the resource type and resource ID of the component's executable code. By convention, this field should be set to the value `kSoundComponentCodeType`:

```
#define kSoundComponentCodeType     'sift'   /*sound component code type*/
```

(You can, however, specify some other resource type if you wish.) The resource ID can be any integer greater than or equal to 128. See the following section for further information about this code resource. The `ResourceSpec` data type has this structure:

```
typedef struct {
    OSType                  resType;
    short                   resID;
} ResourceSpec;
```

The `componentName` field specifies the resource type and resource ID of the resource that contains the component's name. Usually the name is contained in a resource of type `'STR '`. This string should be as short as possible.

The `componentInfo` field specifies the resource type and resource ID of the resource that contains a description of the component. Usually the description is contained in a resource of type `'STR '`.

The `componentIcon` field specifies the resource type and resource ID of the resource that contains an icon for the component. Usually the icon is contained in a resource of type `'ICON'`.

The `cd` field of the `ComponentResource` structure is a **component description record,** which contains additional information about the component. A component description record is defined by the `ComponentDescription` data type.

```
typedef struct {
    OSType                  componentType;
    OSType                  componentSubType;
    OSType                  componentManufacturer;
    unsigned long           componentFlags;
    unsigned long           componentFlagsMask;
} ComponentDescription;
```

For sound components, the `componentType` field must be set to a value recognized by the Sound Manager. Currently, there are five available component types for sound components:

```
#define kSoundComponentType         'sift'   /*utility component*/
#define kMixerType                  'mixr'   /*mixer component*/
#define kSoundHardwareType          'sdev'   /*sound output device component*/
#define kSoundCompressor            'scom'   /*compression component*/
#define kSoundDecompressor          'sdec'   /*decompression component*/
```

In addition, the componentSubType field must be set to a value that indicates the type of audio services your component provides. For example, the Apple-supplied sound output device components have these subtypes:

```
#define kClassicSubType          'clas'   /*Classic hardware*/
#define kASCSubType              'asc '   /*ASC device*/
#define kDSPSubType              'dsp '   /*DSP device*/
```

If you add your own sound output device component, you should define some other subtype.

Note
Apple Computer, Inc., reserves for its own use all types and subtypes composed solely of lowercase letters. ◆

You can assign any value you like to the componentManufacturer field; typically you put the signature of your sound component in this field.

The componentFlags field of the component description for a sound component contains bit flags that encode information about the component. You can use this field to specify that the Component Manager should send your component the kComponentRegisterSelect selector.

```
enum {
    cmpWantsRegisterMessage    = 1L<<31 /*send register request*/
};
```

This bit is most useful for sound output device components, which might need to test for the presence of the appropriate hardware to determine whether to register with the Component Manager. When your component gets the kComponentRegisterSelect selector at system startup time, it should make sure that all the necessary hardware is available. If it isn't available, your component shouldn't register. See "Registering and Opening a Sound Component" beginning on page 281 for more information on opening and registering your sound component.

You also use the componentFlags field of the component description to define the characteristics of your component. For example, you can set a bit in that field to indicate that your sound component can accept stereo sound data. See "Specifying Sound Component Capabilities" on page 276 for more details on specifying the features of your sound component.

You should set the componentFlagsMask field to 0.

Listing 4-1 shows, in Rez format, a component resource for a sample sound output device component named SurfBoard.

4

Sound Components

Listing 4-1   Rez input for a component resource

```
#define kSurfBoardID                      128
#define kSurfBoardSubType                 'SURF'
resource 'thng' (kSurfBoardID, purgeable) {
   'sdev',                      /*component type*/
   kSurfBoardSubType,           /*component subtype*/
   'appl',                      /*component manufacturer*/
   cmpWantsRegisterMessage,     /*component flags*/
   0,                           /*component flags mask*/
   'sift',                      /*component code resource type*/
   kSurfBoardID,                /*component code resource ID*/
   'STR ',                      /*component name resource type*/
   kSurfBoardID,                /*component name resource ID*/
   'STR ',                      /*component info resource type*/
   kSurfBoardID+1,              /*component info resource ID*/
   'ICON',                      /*component icon resource type*/
   kSurfBoardID                 /*component icon resource ID*/
};
```

Your sound component is contained in a resource file. You can assign any type you wish to be the file creator, but the type of the file must be 'thng'. If the sound component contains a 'BNDL' resource, then the file's bundle bit must be set.

Specifying Sound Component Capabilities

As mentioned in the previous section, the componentFlags field of a component description for a sound component contains bit flags that encode information about the component. The high-order 8 bits of that field are reserved for use by the Component Manager. In those 8 bits, you can set the cmpWantsRegisterMessage bit to indicate that the Component Manger should call your component during registration.

The low-order 24 bits of the componentFlags field of a component description are used by the Sound Manager. You'll set some of these bits to define the capabilities of your sound component. You can use the following constants to set specific bits in the componentFlags field.

```
#define k8BitRawIn            (1 << 0)    /*data flags*/
#define k8BitTwosIn           (1 << 1)
#define k16BitIn              (1 << 2)
#define kStereoIn             (1 << 3)
#define k8BitRawOut           (1 << 8)
#define k8BitTwosOut          (1 << 9)
#define k16BitOut             (1 << 10)
#define kStereoOut            (1 << 11)
#define kReverse              (1 << 16)   /*action flags*/
```

```
#define kRateConvert            (1 << 17)
#define kCreateSoundSource      (1 << 18)
#define kHighQuality            (1 << 22)   /*performance flags*/
#define kRealTime               (1 << 23)
```

These constants define four types of information about your sound component: the kind of audio data it can accept as input, the kind of audio data it can produce as output, the actions it can perform on the audio data it's passed, and the performance of your sound component. For example, a utility component that accepts only monaural 8-bit, offset binary data as input and converts it to 16-bit two's complement data might have the value 0x00000801 (that is, k8BitRawIn | k16BitOut) in the componentFlags field.

The Sound Manager also defines a number of masks that you can use to select ranges of bits within the componentFlags field. See "Sound Component Features Flags" on page 291 for complete information on the defined bit constants and masks.

Dispatching to Sound Component-Defined Routines

As explained earlier, the code stored in the sound component should be contained in a resource of type kSoundComponentCodeType. The Component Manager expects the entry point in this resource to be a function with this format:

```
pascal ComponentResult MySurfDispatch (ComponentParameters *params,
                                       SoundComponentGlobalsPtr globals);
```

The Component Manager calls your sound component by passing MySurfDispatch a selector in the params->what field; MySurfDispatch must interpret the selector and possibly dispatch to some other routine in the resource. Your sound component must be able to handle the required selectors, defined by these constants:

```
#define kComponentOpenSelect        -1
#define kComponentCloseSelect       -2
#define kComponentCanDoSelect       -3
#define kComponentVersionSelect     -4
#define kComponentRegisterSelect    -5
#define kComponentTargetSelect      -6
#define kComponentUnregisterSelect  -7
```

Note
For complete details on required component selectors, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.* ◆

In addition, your sound component must be able to respond to component-specific selectors. Some of these selectors must be handled by your component; if your component doesn't implement one of these selectors, it should return the badComponentSelector result code. Other selectors should be delegated up the component chain. This allows the Sound Manager to query a particular component chain by passing a selector to the first component in the chain. If your component does

not implement a delegable selector, it should call the Component Manager routine `DelegateComponentCall` to delegate the selector to its source component. If your sound component does implement a particular delegable selector, it should perform the operation associated with it. The Sound Manager defines a constant to designate the delegable selectors.

```
/*first selector that can be delegated up the chain*/
#define kDelegatedSoundComponentSelectors        0x0100
```

The Sound Manager can pass these selectors to your sound component:

```
enum {
   /*the following calls cannot be delegated*/
   kSoundComponentInitOutputDeviceSelect      = 1,
   kSoundComponentSetSourceSelect,
   kSoundComponentGetSourceSelect,
   kSoundComponentGetSourceDataSelect,
   kSoundComponentSetOutputSelect,
   /*the following calls can be delegated*/
   kSoundComponentAddSourceSelect = kDelegatedSoundComponentSelectors + 1,
   kSoundComponentRemoveSourceSelect,
   kSoundComponentGetInfoSelect,
   kSoundComponentSetInfoSelect,
   kSoundComponentStartSourceSelect,
   kSoundComponentStopSourceSelect,
   kSoundComponentPauseSourceSelect,
   kSoundComponentPlaySourceBufferSelect
};
```

You can respond to these selectors by calling the Component Manager routine `CallComponentFunctionWithStorage` or by delegating the selector to your component's source component. Listing 4-2 illustrates how to define a sound component entry point routine.

--------------- **Listing 4-2**   Handling Component Manager selectors

```
pascal ComponentResult MySurfDispatch (ComponentParameters *params,
                                          SoundComponentGlobalsPtr globals)
{
   ComponentRoutine      myRoutine;
   ComponentResult       myResult;

   /*Get address of component-defined routine.*/
   myRoutine = MyGetComponentRoutine(params->what);
```

```
  if (myRoutine == nil)                    /*selector not implemented*/
     myResult = badComponentSelector;
  else if (myRoutine == kDelegateCall)     /*selector should be delegated*/
     myResult = DelegateComponentCall(params, globals->sourceComponent);
  else
     myResult = CallComponentFunctionWithStorage((Handle) globals, params,
                                         (ComponentRoutine) myRoutine);
  return (myResult);
}
```

As you can see, the `MySurfDispatch` function defined in Listing 4-2 simply retrieves the address of the appropriate component-defined routine, as determined by the `params->what` field. If the routine `MyGetComponentRoutine` returns `nil`, then `MySurfDispatch` itself returns the `badComponentSelector` result code. Otherwise, if the selector should be delegated, `MySurfDispatch` calls `DelegateComponentCall` to do so. Finally, if the selector hasn't yet been handled, the appropriate component-defined routine is executed via `CallComponentFunctionWithStorage`.

Listing 4-3 defines the function `MyGetComponentRoutine`.

Listing 4-3   Finding the address of a component-defined routine

```
ComponentRoutine MyGetComponentRoutine (short selector)
{
   void                 *myRoutine;

   if (selector < 0)
      switch (selector)          /*required component selectors*/
      {
         case kComponentRegisterSelect:
            myRoutine = MyRegisterSoundComponent;
            break;
         case kComponentVersionSelect:
            myRoutine = MySoundComponentVersion;
            break;
         case kComponentCanDoSelect:
            myRoutine = MySoundComponentCanDo;
            break;
         case kComponentCloseSelect:
            myRoutine = MyCloseSoundComponent;
            break;
         case kComponentOpenSelect:
            myRoutine = MyOpenSoundComponent;
            break;
```

4

Sound Components

```
            default:
               myRoutine = nil;      /*unknown selector, so fail*/
               break;
      }
   else if (selector < kDelegatedSoundComponentSelectors)
                           /*selectors that can't be delegated*/
      switch (selector)
      {
         case kSoundComponentInitOutputDeviceSelect:
            myRoutine = MySoundComponentInitOutputDevice;
            break;


         case kSoundComponentSetSourceSelect:
         case kSoundComponentGetSourceSelect:
         case kSoundComponentGetSourceDataSelect:
         case kSoundComponentSetOutputSelect:
         default:
            myRoutine = nil;      /*unknown selector, so fail*/
            break;
      }
   else                    /*selectors that can be delegated*/
      switch (selector)
      {
         case kSoundComponentStartSourceSelect:
            myRoutine = MySoundComponentStartSource;
            break;
         case kSoundComponentPlaySourceBufferSelect:
            myRoutine = MySoundComponentPlaySourceBuffer;
            break;
         case kSoundComponentGetInfoSelect:
            myRoutine = MySoundComponentGetInfo;
            break;
         case kSoundComponentSetInfoSelect:
            myRoutine = MySoundComponentSetInfo;
            break;
         case kSoundComponentAddSourceSelect:
         case kSoundComponentRemoveSourceSelect:
         case kSoundComponentStopSourceSelect:
         case kSoundComponentPauseSourceSelect:
         default:
            myRoutine = kDelegateCall;             /*delegate it*/
            break;
      }
```

```
    return (myRoutine);
}
```

In all likelihood, your component is loaded into the system heap, although it might be loaded into an application heap if memory is low in the system heap. You can call the Component Manager function `GetComponentInstanceA5` to determine the A5 value of the current application. If this function returns 0, your component is in the system heap; otherwise, your component is in an application's heap. Its location might affect how you allocate memory. For example, calling the `MoveHHi` routine on handles in the system heap has no result. Thus, you should either call the `ReserveMemSys` routine before calling `NewHandleSys` (so that the handle is allocated as low in the system heap as possible) or else just allocate a nonrelocatable block by calling the `NewPtrSys` routine.

If you need to access resources that are stored in your sound component, you can use `OpenComponentResFile` and `CloseComponentResFile`. `OpenComponentResFile` requires the `ComponentInstance` parameter supplied to your routine. You should not call Resource Manager routines such as `OpenResFile` or `CloseResFile`.

▲ **WARNING**
Do not leave any resource files open when your sound component is closed. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. ▲

The following sections illustrate how to define some of the sound component functions.

Registering and Opening a Sound Component

The Component Manager sends your component the `kComponentRegisterSelect` selector, usually at system startup time, to allow your component to determine whether it wants to register itself with the Component Manager. Utility components should always register themselves, so that the capabilities they provide will be available when needed. Sound output device components, however, should first check to see whether any necessary hardware is available before registering themselves. If the hardware they drive isn't available, there is no point in registering with the Component Manager.

The Component Manager sends your component the `kComponentOpenSelect` selector whenever the Sound Manager wants to open a connection to your component. In general, a sound output device component has only one connection made to it. A utility component, however, might have several instances, if the capabilities it provides are needed by more than one sound component chain. Your component should do as little as possible when opening up. It should allocate whatever global storage it needs to manage the connection and call `SetComponentInstanceStorage` so that the Component Manager can remember the location of that storage and pass it to all other component-defined routines.

As noted in the previous section, your component is probably loaded into the system heap. If so, you should also allocate any global storage in the system heap. If memory is tight, however, your component might be loaded into an application's heap (namely,

the heap of the first application that plays sound). In that case, you should allocate any global variables you need in that heap. The Sound Manager ensures that other applications will not try to play sound while the component is in this application heap.

IMPORTANT

Your component is always sent the `kComponentOpenSelect` component selector before it is sent the `kComponentRegisterSelect` selector. As a result, you should not attempt to initialize or configure any associated hardware in response to `kComponentOpenSelect`. ▲

The Sound Manager sends the `kSoundComponentInitOutputDeviceSelect` selector specifically to allow a sound output device component to perform any hardware-related operations. Your component should initialize the hardware to some reasonable default values, create the Apple Mixer, and allocate any other memory that might be needed. Listing 4-4 shows one way to respond to the `kSoundComponentInitOutputDeviceSelect` selector.

Listing 4-4   Initializing an output device

```
static pascal ComponentResult MySoundComponentInitOutputDevice
                    (SoundComponentGlobalsPtr globals, long actions)
{
#pragma unused (actions)
   ComponentResult         myResult;

   /*Make sure we got our globals.*/
   if (globals->hwGlobals == nil)
      return (notEnoughHardwareErr);

   /*Set up the hardware.*/
   myResult = MySetupHardware(globals);
   if (myResult != noErr)
      return (myResult);

   /*Create an Apple Mixer.*/
   myResult = OpenMixerSoundComponent(&globals->thisComp, 0,
                                  &globals->sourceComponent);

   return (myResult);
}
```

The `MySoundComponentInitOutputDevice` function defined in Listing 4-4 simply retrieves the location of its global variables, configures the hardware by calling the `MySetupHardware` function, and then calls `OpenMixerSoundComponent` to create an instance of the Apple Mixer.

Finding and Changing Component Capabilities

All sound components take a stream of input data and produce a (usually different) stream of output data. The Sound Manager needs to know what operations your component can perform, so that it knows what other sound components might need to be linked together to play a particular sound on the available sound output device. It calls your component's SoundComponentGetInfo and SoundComponentSetInfo functions to get and set information about the capabilities and current settings of your sound component.

To specify the kind of information it wants to get or set, the Sound Manager passes your component a **sound component information selector.** If your component does not support a particular selector, if should pass the selector to the specified sound source. If your component does support the selector, it should either return the desired information directly or alter its settings as requested.

The sound component information selectors can specify any of a large number of audio capabilities or component settings. For example, the selector siRateMultiplier is passed to get or set the current output sample rate multiplier value.

Note
The Sound Manager uses many of the sound input device information selectors defined by the Sound Input Manager for communicating with sound input devices. See "Sound Input Manager" in this book for a description of the sound input device information selectors. A complete list of all sound component information selectors is provided in "Sound Component Information Selectors" beginning on page 287. ◆

Your component's SoundComponentGetInfo function has the following declaration:

```
pascal ComponentResult SoundComponentGetInfo (ComponentInstance ti,
                                    SoundSource sourceID, OSType selector,
                                    void *infoPtr);
```

The sound component information selector is passed in the selector parameter. The sound source is identified by the source ID passed in the sourceID parameter. The infoPtr parameter specifies the location in memory of the information returned by SoundComponentGetInfo. If the information to be returned occupies four bytes or fewer, you can simply return the information in the location pointed to by that parameter. Otherwise, you should pass back in the infoPtr parameter a pointer to a record of type SoundInfoList, which contains an integer and a handle to an array of data items. In the second case, you'll need to allocate memory to hold the information you need to pass back. Listing 4-5 defines a component's SoundComponentGetInfo routine. It returns information to the Sound Manager about its capabilities and current settings.

Listing 4-5   Getting sound component information

```
static pascal ComponentResult MySoundComponentGetInfo
                    (SoundComponentGlobalsPtr globals, SoundSource sourceID,
                      OSType selector, void *infoPtr)
{
   HandleListPtr        listPtr;
   short                *sp, i;
   UnsignedFixed        *lp;
   Handle               h;
   HardwareGlobalsPtr   hwGlobals = globals->hwGlobals;
   ComponentResult      result = noErr;

   /*Make sure we got our global variables.*/
   if (hwGlobals == nil)
      return (notEnoughHardwareErr);

   switch (selector)
   {
      case siSampleSize:                      /*return current sample size*/
         *((short *) infoPtr) = hwGlobals->sampleSize;
         break;

      case siSampleSizeAvailable:         /*return sample sizes available*/
         h = NewHandle(sizeof(short) * kSampleSizesCount);
         if (h == nil)
            return (MemError());

         listPtr = (HandleListPtr) infoPtr;
         listPtr->count = 0;                 /*num. sample sizes in handle*/
         listPtr->handle = h;                /*handle to be returned*/

         sp = (short *) *h;                  /*store sample sizes in handle*/

         for (i = 0; i < kSampleSizesCount; ++i)
            if (hwGlobals->sampleSizesActive[i])
            {
               listPtr->count++;
               *sp++ = hwGlobals->sampleSizes[i];
            }
         break;

      case siSampleRate:                      /*return current sample rate*/
         *((Fixed *) infoPtr) = hwGlobals->sampleRate;
```

```
    break;

case siSampleRateAvailable:           /*return sample rates available*/
   h = NewHandle(sizeof(UnsignedFixed) * kSampleRatesCount);
   if (h == nil)
       return (MemError());

   listPtr = (HandleListPtr) infoPtr;
   listPtr->count = 0;                /*num. sample rates in handle*/
   listPtr->handle = h;               /*handle to be returned*/

   lp = (UnsignedFixed *) *h;

   /*If the hardware can support a range of sample rate values,
     the list count should be set to 0 and the minimum and maximum
     sample rate values should be stored in the handle.*/
   if (hwGlobals->supportsRateRange)
   {
       *lp++ = hwGlobals->sampleRateMin;
       *lp++ = hwGlobals->sampleRateMax;
   }

   /*If the hardware supports a limited set of sample rates,
     the list count should be set to the number of sample rates
     and this list of rates should be stored in the handle.*/
   else
   {
       for (i = 0; i < kSampleRatesCount; ++i)
          if (hwGlobals->sampleRatesActive[i])
          {
             listPtr->count++;
             *lp++ = hwGlobals->sampleRates[i];
          }
   }
   break;

case siNumberChannels:                /*return current num. channels*/
   *((short *) infoPtr) = hwGlobals->numChannels;
   break;

case siChannelAvailable:              /*return channels available*/
   h = NewHandle(sizeof(short) * kChannelsCount);
   if (h == nil)
```

```
        return (MemError());

    listPtr = (HandleListPtr) infoPtr;
    listPtr->count = 0;                /*num. channels in handle*/
    listPtr->handle = h;               /*handle to be returned*/

    sp = (short *) *h;                 /*store channels in handle*/

    for (i = 0; i < kChannelsCount; ++i)
        if (hwGlobals->channelsActive[i])
        {
            listPtr->count++;
            *sp++ = hwGlobals->channels[i];
        }
    break;

case siHardwareVolume:
    *((long *)infoPtr) = hwGlobals->volume;
    break;

/*If you do not handle a selector, delegate it up the chain.*/
default:
    result = SoundComponentGetInfo(globals->sourceComponent, sourceID,
                                selector, infoPtr);
    break;
    }
    return (result);
}
```

You can define your `MySoundComponentSetInfo` routine in an exactly similar fashion.

Sound Components Reference

This section describes the constants, data structures, and routines you can use to write a sound component. It also describes the routines that your sound component should call in response to a sound component selector. See "Writing a Sound Component" on page 273 for information on creating a component that contains these component-defined routines.

Constants

This section provides details on the constants defined by the Sound Manager for use with sound components. You'll use these constants to

■ determine the kind of information the Sound Manager wants your sound component to return to it or settings it wants your sound component to change

■ define the format of the audio data your sound component is currently producing

■ specify the action flags for the `SoundComponentPlaySourceBuffer` function

■ specify the format of the data your sound output device component expects to receive

Sound Component Information Selectors

The Sound Manager calls your sound component's `SoundComponentGetInfo` and `SoundComponentSetInfo` functions to determine the capabilities of your component and to change those capabilities. It passes those functions a sound component information selector in the function's `selector` parameter to specify the type of information it wants to get or set. The available sound component information selectors are defined by constants.

Note
Most of these selectors can be passed to both
`SoundComponentGetInfo` and `SoundComponentSetInfo`.
Some of them, however, can be sent to only one or the other. ◆

```
#define siChannelAvailable        'chav'   /*number of channels available*/
#define siCompressionAvailable    'cmav'   /*compression types available*/
#define siCompressionFactor       'cmfa'   /*current compression factor*/
#define siCompressionType         'comp'   /*current compression type*/
#define siHardwareMute            'hmut'   /*current hardware mute state*/
#define siHardwareVolume          'hvol'   /*current hardware volume*/
#define siHardwareVolumeSteps     'hstp'   /*number of hardware volume steps*/
#define siHeadphoneMute           'pmut'   /*current headphone mute state*/
#define siHeadphoneVolume         'pvol'   /*current headphone volume*/
#define siHeadphoneVolumeSteps    'hdst'   /*num. of headphone volume steps*/
#define siNumberChannels          'chan'   /*current number of channels*/
#define siQuality                 'qual'   /*current quality*/
#define siRateMultiplier          'rmul'   /*current rate multiplier*/
#define siSampleRate              'srat'   /*current sample rate*/
#define siSampleRateAvailable     'srav'   /*sample rates available*/
#define siSampleSize              'ssiz'   /*current sample size*/
#define siSampleSizeAvailable     'ssav'   /*sample sizes available*/
#define siSpeakerMute             'smut'   /*current speaker mute*/
#define siSpeakerVolume           'svol'   /*current speaker volume*/
#define siVolume                  'volu'   /*current volume setting*/
```

Constant descriptions

siChannelAvailable

Get the maximum number of channels this sound component can manage, as well as the channels themselves. The `infoPtr` parameter points to a record of type `SoundInfoList`, which contains an integer (the number of available channels) and a handle to an array of integers (which represent the channel numbers themselves).

siCompressionAvailable

Get the number and list of compression types this sound component can manage. The `infoPtr` parameter points to a record of type `SoundInfoList`, which contains the number of compression types, followed by a handle that references a list of compression types, each of type `OSType`.

siCompressionFactor

Get information about the current compression type. The `infoData` parameter points to a compression information record (see page 297).

siCompressionType

Get or set the current compression type. The `infoPtr` parameter points to a buffer of type `OSType`, which is the compression type.

siHardwareMute

Get or set the current mute state of the audio hardware. A value of 0 indicates that the hardware is not muted, and a value of 1 indicates that the hardware is muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated hardware can be muted.

siHardwareVolume

Get or set the current volume level of all sounds produced on the sound output device. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the output device, whereas the `siVolume` selector applies to the volume of a specific sound channel and its component chain. If a sound output device supports more than one output port (for example, both headphones and speakers), the `siHardwareVolume` selector applies to all those ports.

siHardwareVolumeSteps

Get the number of audible volume levels supported by the audio hardware. If the device supports a range of volume levels (for example, 0x000 to 0x1000), you should return only the number of levels that are audible. The Sound Manager uses this information to handle the volume slider in the Alert Sounds control panel.

siHeadphoneMute

Get or set the current mute state of the headphone. A value of 0 indicates that the headphone is not muted, and a value of 1 indicates that the headphone is muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated headphone can be muted.

siHeadphoneVolume

Get or set the current volume level of all sounds produced on the headphone. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the headphones.

siHeadphoneVolumeSteps

Get the number of audible volume levels supported by the headphones. If the headphones support a range of volume levels (for example, 0x000 to 0x1000), you should return only the number of levels that are audible.

siNumberChannels

Get or set the current number of audio channels currently being managed by the sound component. The `infoPtr` parameter points to an integer, which is the number of channels. For example, for stereo sounds, this integer should be 2.

siQuality   Get or set the current quality setting for the sound component. The `infoPtr` parameter points to a 32-bit value, which typically determines how much processing should be applied to the audio data stream.

siRateMultiplier

Get or set the current rate multiplier for the sound component. The `infoPtr` parameter points to a buffer of type `UnsignedFixed`, which is the multiplier to be applied to the playback rate of the sound, independent of the base sample rate of the sound. For example, if the current rate multiplier is 2.0, the sound is played back at twice the speed specified in the `sampleRate` field of the sound component data record.

siSampleRate   Get or set the current sample rate of the data being output by the sound component. The `infoPtr` parameter points to a buffer of type `UnsignedFixed`, which is the sample rate.

siSampleRateAvailable

Get the range of sample rates this sound component can handle. The `infoPtr` parameter points to a record of type `SoundInfoList`, which is the number of sample rates the component supports, followed by a handle to a list of sample rates, each of type `UnsignedFixed`. The sample rates can be in the range 0 to 65535.65535. If the number of sample rates is 0, then the first two

sample rates in the list define the lowest and highest values in a continuous range of sample rates.

siSampleSize   Get or set the current sample size of the audio data being output by the sound component. The `infoPtr` parameter points to an integer, which is the sample size in bits.

siSampleSizeAvailable

Get the range of sample sizes this sound component can handle. The `infoPtr` parameter points to a record of type `SoundInfoList`, which is the number of sample sizes the sound component supports, followed by a handle. The handle references a list of sample sizes, each of type `Integer`. Sample sizes are specified in bits.

siSpeakerMute

Get or set the current mute state of the speakers. A value of 0 indicates that the speakers are not muted, and a value of 1 indicates that the speakers are muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated speakers can be muted.

siSpeakerVolume

Get or set the current volume level of all sounds produced on the speakers. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the speakers.

siVolume   Get or set the current volume level of the sound component. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of a specific sound channel and its component chain, while the `siHardwareVolume` selector applies to the volume of the output device.

Audio Data Types

You can use the following constants to define the format of the audio data your sound component is currently producing. You can also define additional data types to denote your own compression schemes. You pass these constants in the `format` field of a sound component data record.

```
#define kOffsetBinary          'raw '
#define kTwosComplement        'twos'
#define kMACE3Compression      'MAC3'
#define kMACE6Compression      'MAC6'
```

Constant descriptions

kOffsetBinary   The data is noncompressed samples in offset binary format (that is, values range from 0 to 255).

kTwosComplement
                The data is noncompressed samples in two's complement format (that is, values range from –128 to 128).

kMACE3Compression
                The data is compressed using MACE 3:1 compression.

kMACE6Compression
                The data is compressed using MACE 6:1 compression.

Sound Component Features Flags

You can use the following constants to define features of your sound component. You use some combination of these constants to set bits in the componentFlags field of a component description record, which is contained in a 'thng' resource. These bits represent the kind of data your component can receive as input, the kind of data your component can produce as output, the operations your component can perform, and the performance of your component.

```
#define k8BitRawIn             (1 << 0)    /*data flags*/
#define k8BitTwosIn            (1 << 1)
#define k16BitIn               (1 << 2)
#define kStereoIn              (1 << 3)
#define k8BitRawOut            (1 << 8)
#define k8BitTwosOut           (1 << 9)
#define k16BitOut              (1 << 10)
#define kStereoOut             (1 << 11)
#define kReverse               (1 << 16)   /*action flags*/
#define kRateConvert           (1 << 17)
#define kCreateSoundSource     (1 << 18)
#define kHighQuality           (1 << 22)   /*performance flags*/
#define kRealTime              (1 << 23)
```

Constant descriptions

k8BitRawIn   The component can accept 8 bit offset binary data as input.

k8BitTwosIn   The component can accept 8 bit two's complement data as input.

k16BitIn   The component can accept 16 bit data as input. 16 bit data is always in two's complement format.

kStereoIn   The component can accept stereo data as input.

k8BitRawOut	The component can produce 8 bit offset binary data as output.
k8BitTwosOut	The component can produce 8 bit two's complement data as output.
k16BitOut	The component can produce 16 bit data as output. 16 bit data is always in two's complement format.
kStereoOut	The component can produce stereo data as output.
kReverse	The component can accept reversed audio data.
kRateConvert	The component can convert sample rates.
kCreateSoundSource	
	The component can create sound sources.
kHighQuality	The component can produce high quality output.
kRealTime	The component can operate in real time.

Action Flags

You can use constants to specify the action flags in the `actions` parameter of the `SoundComponentPlaySourceBuffer` function. See page 314 for information about this function.

```
#define kSourcePaused            (1 << 0)
#define kPassThrough             (1 << 16)
#define kNoSoundComponentChain   (1 << 17)
```

Constant descriptions

kSourcePaused	If this bit is set, the component chain is configured to play the specified sound but the playback is initially paused. In this case, your `SoundComponentStartSource` function must be called to begin playback. If this bit is clear, the playback begins immediately once the component chain is set up and configured.
kPassThrough	If this bit is set, the Sound Manager passes all data through to the sound output device component unmodified. A sound output device component that can handle any sample rate and sound format described in a sound parameter block should set this bit.
kNoSoundComponentChain	
	If this bit is set, the Sound Manager does not construct a component chain for processing the sound data.

Data Format Flags

You can use constants to set or clear flag bits in the `outputFlags` parameter passed to the `OpenMixerSoundComponent` routine. These flags specify the format of the data your sound output device component expects to receive. See page 298 for information about the `OpenMixerSoundComponent` function.

IMPORTANT

Most of these flags are ignored unless the `kNoMixing` flag is set, because
a sound output device component cannot perform data modifications
such as sample rate conversion or sample size conversion unless it is also
able to mix sound sources. ▲

```
#define kNoMixing               (1 << 0)    /*don't mix sources*/
#define kNoSampleRateConversion (1 << 1)    /*don't convert sample rate*/
#define kNoSampleSizeConversion (1 << 2)    /*don't convert sample size*/
#define kNoSampleFormatConversion         \
                                (1 << 3)    /*don't convert sample format*/
#define kNoChannelConversion    (1 << 4)    /*don't convert stereo/mono*/
#define kNoDecompression        (1 << 5)    /*don't decompress*/
#define kNoVolumeConversion     (1 << 6)    /*don't apply volume*/
#define kNoRealtimeProcessing   (1 << 7)    /*don't run at interrupt time*/
```

Constant descriptions

kNoMixing   If this bit is set, the Apple Mixer does not mix audio data sources.

kNoSampleRateConversion

                If this bit is set, the sound component chain does not perform
sample rate conversion (for example, converting 11 kHz data to
22 kHz data).

kNoSampleSizeConversion

                If this bit is set, the sound component chain does not perform
sample size conversion (for example, converting 8-bit data to
16-bit data).

kNoSampleFormatConversion

                If this bit is set, the sound component chain does not convert
between sample formats (for example, converting from two's
complement data to offset binary data). Most sound output devices
on Macintosh computers accept only 8-bit offset binary data, which
is therefore the default type of data produced by the Apple Mixer.
If your output device can handle either offset binary or two's
complement data, you should set this flag. Note that 16-bit data
is always in two's complement format.

kNoChannelConversion

                If this bit is set, the sound component chain does not convert
channels (for example, converting monophonic channels to stereo or
stereo channels to monophonic).

kNoDecompression

                If this bit is set, the sound component chain does not decompress
audio data. If your output device can decompress data, you should
set this flag.

kNoVolumeConversion

                If this bit is set, the sound component chain does not convert
volumes.

kNoRealtimeProcessing
                    If this bit is set, the sound component chain does not do any
                    processing at interrupt time.

Data Structures

This section describes the data structures you need to use when writing a sound component.

Sound Component Data Records

The flow of data from one sound component to another is managed using a sound component data record. This record indicates to other sound components the format of the data that a particular component is generating, together with the location and length of the buffer containing that data. This allows other sound components to access data from that component as needed. A sound component data record is defined by the `SoundComponentData` data type.

```
typedef struct {
   long           flags;              /*sound component flags*/
   OSType         format;             /*data format*/
   short          numChannels;        /*number of channels in data*/
   short          sampleSize;         /*size of a sample*/
   UnsignedFixed  sampleRate;         /*sample rate*/
   long           sampleCount;        /*number of samples in buffer*/
   Byte           *buffer;            /*location of data*/
   long           reserved;           /*reserved*/
} SoundComponentData, *SoundComponentDataPtr;
```

Field descriptions

flags   A set of bit flags whose meanings are specific to a particular sound
                component.

format   The format of the data a sound component is producing. The
                following formats are defined by Apple:

```
#define kOffsetBinary          'raw '
#define kTwosComplement        'twos'
#define kMACE3Compression      'MAC3'
#define kMACE6Compression      'MAC6'
```

                See "Audio Data Types" on page 290 for a description of these
                formats. You can define additional format types, which are currently
                assumed to be the types of proprietary compression algorithms.

numChannels   The number of channels of sound in the output data stream. If this
                field contains the value 1, the data is monophonic. If this field

contains 2, the data is stereophonic. Stereo data is stored as interleaved samples, in a left-to-right ordering.

sampleSize	The size, in bits, of each sample in the output data stream. Typically this field contains the values 8 or 16. For compressed sound data, this field indicates the size of the samples after the data has been expanded.
sampleRate	The sample rate for the audio data. The sample rate is expressed as an unsigned, fixed-point number in the range 0 to 65536.0 samples per second.
sampleCount	The number of samples in the buffer pointed to by the `buffer` field. For compressed sounds, this field indicates the number of compressed samples in the sound, not the size of the buffer.
buffer	The location of the buffer that contains the sound data.
reserved	Reserved for future use. You should set this field to 0.

Sound Parameter Blocks

The Sound Manager passes a component's `SoundComponentPlaySourceBuffer` function a **sound parameter block** that describes the source data to be modified or sent to a sound output device. A sound parameter block is defined by the `SoundParamBlock` data type.

```
struct SoundParamBlock {
    long                recordSize;     /*size of this record in bytes*/
    SoundComponentData  desc;           /*description of sound buffer*/
    Fixed               rateMultiplier;/*rate multiplier*/
    short               leftVolume;     /*volume on left channel*/
    short               rightVolume;    /*volume on right channel*/
    long                quality;        /*quality*/
    ComponentInstance   filter;         /*filter*/
    SoundParamProcPtr   moreRtn;        /*routine to call to get more data*/
    SoundParamProcPtr   completionRtn;  /*buffer complete routine*/
    long                refCon;         /*user refcon*/
    short               result;         /*result*/
};
typedef struct SoundParamBlock SoundParamBlock;
typedef SoundParamBlock *SoundParamBlockPtr;
```

Field descriptions

recordSize	The length, in bytes, of the sound parameter block.
desc	A sound component data record that describes the format, size, and location of the sound data. See "Sound Component Data Records" on page 294 for a description of the sound component data record.
rateMultiplier	
	A multiplier to be applied to the playback rate of the sound. This field contains an unsigned fixed-point number. If, for example, this

4

Sound Components

field has the value 2.0, the sound is played back at twice the rate specified in the `sampleRate` field of the sound component data record contained in the `desc` field.

`leftVolume`	The playback volume for the left channel. You specify a volume with 16-bit value, where 0 (hexadecimal 0x0000) represents no volume and 256 (hexadecimal 0x0100) represents full volume. You can overdrive a channel's volume by passing volume levels greater than 0x0100.
`rightVolume`	The playback volume for the right channel. You specify a volume with 16-bit value, where 0 (hexadecimal 0x0000) represents no volume and 256 (hexadecimal 0x0100) represents full volume. You can overdrive a channel's volume by passing volume levels greater than 0x0100.
`quality`	The level of quality for the sound. This value usually determines how much processing should be applied during audio data processing (such as rate conversion and decompression) to increase the output quality of the sound.
`filter`	Reserved for future use. You should set this field to `nil`.
`moreRtn`	A pointer to a callback routine that is called to retrieve another buffer of audio data. This field is used internally by the Sound Manager.
`completionRtn`	A pointer to a callback routine that is called when the sound has finished playing. This field is used internally by the Sound Manager.
`refCon`	A value that is to be passed to the callback routines specified in the `moreRtn` and `completionRtn` fields. You can use this field to pass information (for example, the address of a structure) to a callback routine.
`result`	The status of the sound that is playing. The value 1 indicates that the sound is currently playing. The value 0 indicates that the sound has finished playing. Any negative value indicates that some error has occurred.

Sound Information Lists

The `SoundComponentGetInfo` and `SoundComponentSetInfo` functions access information about a sound component using a **sound information list,** which is defined by the `SoundInfoList` data type.

```
typedef struct {
    short          count;
    Handle         handle;
} SoundInfoList, *SoundInfoListPtr;
```

Field descriptions

`count`	The number of elements in the array referenced by the `handle` field.
`handle`	A handle to an array of data elements. The type of these data elements depends on the kind of information requested, which

is determined by the `selector` parameter passed to `SoundComponentGetInfo` or `SoundComponentSetInfo`. See "Sound Component Information Selectors" beginning on page 287 for information about the available information selectors.

Compression Information Records

When the Sound Manager calls your `SoundComponentGetInfo` routine with the `siCompressionFactor` selector, you need to return a pointer to a **compression information record,** which is defined by the `CompressionInfo` data type.

```
typedef struct {
    long            recordSize;
    OSType          format;
    short           compressionID;
    short           samplesPerPacket;
    short           bytesPerPacket;
    short           bytesPerFrame;
    short           bytesPerSample;
    short           futureUse1;
} CompressionInfo, *CompressionInfoPtr, **CompressionInfoHandle;
```

Field descriptions

`recordSize`	The size of this compression information record.
`format`	The compression format.
`compressionID`	The compression ID.
`samplesPerPacket`	
	The number of samples in each packet.
`bytesPerPacket`	
	The number of bytes in each packet.
`bytesPerFrame`	
	The number of bytes in each frame.
`bytesPerSample`	
	The number of bytes in each sample.
`futureUse1`	Reserved for use by Apple Computer, Inc. You should set this field to 0.

Sound Manager Utilities

This section describes several utility routines provided by the Sound Manager that are intended for use only by sound components. You can use these routines to

- open and close the Apple Mixer component

- save and restore a user's preference settings for a sound component

Note
For a description of the routines that a sound component must implement, see "Sound Component-Defined Routines" on page 301. ◆

Opening and Closing the Apple Mixer Component

A sound output device component needs to open and close one or more instances of the Apple Mixer component.

OpenMixerSoundComponent

A sound output device component can use the `OpenMixerSoundComponent` function to open and connect itself to the Apple Mixer component.

```
pascal OSErr OpenMixerSoundComponent
                        (SoundComponentDataPtr outputDescription,
                        long outputFlags,
                        ComponentInstance *mixerComponent);
```

outputDescription
    A description of the data format your sound output device is expecting to receive.

outputFlags
    A set of 32 bit flags that provide additional information about the data format your output device is expecting to receive. See "Data Format Flags" beginning on page 292 for a description of the constants you can use to select bits in this parameter.

mixerComponent
    The component instance of the Apple Mixer component. You need this instance to call the `SoundComponentGetSourceData` and `CloseMixerSoundComponent` functions.

DESCRIPTION

The `OpenMixerSoundComponent` function opens the standard Apple Mixer component and creates a connection between your sound output device component and the Apple Mixer. If your output device can perform specific operations on the stream of audio data, such as channel mixing and rate conversion, it should call `OpenMixerSoundComponent` as many times as are necessary to create a unique component chain for each sound source. If, on the other hand, your output device does not perform channel mixing, it should call `OpenMixerSoundComponent` only once, from its `SoundComponentInitOutputDevice` function. This opens a single instance of the Apple Mixer component, which in turn manages all the available sound sources.

Your component specifies the format of the data it can handle by filling in a sound component data record and passing its address in the outputDescription parameter. The sound component data record specifies the data format as well as the sample rate and sample size expected by the output device component. If these specifications are sufficient to determine the kind of data your component can handle, you should pass the value 0 in the outputFlags parameter. Otherwise, you can set flags in the outputFlags parameter to select certain kinds of input data. For example, you can set the kNoChannelConversion flag to prevent the component chain from converting monophonic sound to stereo sound, or stereo sound to monophonic sound. See "Data Format Flags" beginning on page 292 for a description of the constants you can use to select bits in the outputFlags parameter.

SPECIAL CONSIDERATIONS

The OpenMixerSoundComponent function is available only in versions 3.0 and later of the Sound Manager. It should be called only by sound output device components.

CloseMixerSoundComponent

A sound output device component can use the CloseMixerSoundComponent function to close the Apple Mixer.

```pascal
pascal OSErr CloseMixerSoundComponent (ComponentInstance ci);
```

ci   The component instance of the Apple Mixer component.

DESCRIPTION

The CloseMixerSoundComponent function closes the Apple Mixer component instance specified by the ci parameter. Your output device component should call this function when it is being closed.

SPECIAL CONSIDERATIONS

The CloseMixerSoundComponent function is available only in versions 3.0 and later of the Sound Manager. It should be called only by sound output device components.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	Invalid component ID

4

Sound Components

Saving and Restoring Sound Component Preferences

A sound component can use the `SetSoundPreference` and `GetSoundPreference` functions to save and restore a user's preference settings.

SetSoundPreference

A sound component can use the `SetSoundPreference` function to have the Sound Manager store a block of preferences data in a resource file. You're most likely to use this function in a sound output device component, although other types of sound components can use it also.

```
pascal OSErr SetSoundPreference (OSType type, Str255 name,
                                    Handle settings);
```

type   The resource type to be used to create the preferences resource.

name   The resource name to be used to create the preferences resource.

settings   A handle to the data to be stored in the preferences resource.

DESCRIPTION

The `SetSoundPreference` function causes the Sound Manager to attempt to create a new resource that contains preferences data for your sound component. You can use this function to maintain a structure of any format across subsequent startups of the machine. You'll retrieve the preferences data by calling the `GetSoundPreference` function. The data is stored in a resource with the specified type and name in a resource file in the Preferences folder in the System Folder. In general, the resource type and name should be the same as the sound component subtype and name.

The `settings` parameter is a handle to the preferences data you want to store. It is the responsibility of your component to allocate and initialize the block of data referenced by that handle. The Sound Manager copies the handle's data into a resource in the appropriate location. Your sound component should dispose of the handle when `SetSoundPreference` returns.

The format of the block of preferences data referenced by the `settings` parameter is defined by your sound component. It is recommended that you include a field specifying the version of the data format; this allows you to modify the format of the block of data while remaining compatible with previous formats you might have defined.

SPECIAL CONSIDERATIONS

The `SetSoundPreference` function is available only in versions 3.0 and later of the Sound Manager.

GetSoundPreference

A sound component can use the `GetSoundPreference` function to have the Sound Manager read a block of preferences data from a resource file. You'll use it to retrieve a block of preferences data you previously saved by calling `SetSoundPreference`.

```
pascal OSErr GetSoundPreference (OSType type, Str255 name,
                                   Handle settings);
```

type   The resource type of the preferences resource.

name   The resource name of the preferences resource.

settings A handle to the data in the preferences resource.

DESCRIPTION

The `GetSoundPreference` function retrieves the block of preferences data you previously stored in a resource by calling the `SetSoundPreference` function. It is the responsibility of your component to allocate the block of data referenced by the `settings` handle. The Sound Manager resizes the handle (if necessary) and fills it with data from the resource with the specified type and name. Your sound component should dispose of the handle once it's finished reading the data from it. You can determine the size of the handle returned by the Sound Manager by calling the Memory Manager's `GetHandleSize` function.

SPECIAL CONSIDERATIONS

The `GetSoundPreference` function is available only in versions 3.0 and later of the Sound Manager.

Sound Component-Defined Routines

This section describes the routines you need to define in order to write a sound component. You need to write routines to

■ load, configure, and unload your sound component

■ add and remove audio sources

■ read and set component settings

■ control and process audio data

Some of these routines are optional for some types of sound components. All routines return result codes. If they succeed, they should return `noErr`. To simplify dispatching, the Component Manager requires these routines to return a value of type `ComponentResult`.

See "Writing a Sound Component" beginning on page 273 for a description of how you call these routines from within a sound component. See "Sound Manager Utilities"

beginning on page 297 for a description of some Sound Manager utility routines you can use in a sound component.

Managing Sound Components

To write a sound component, you might need to define routines that manage the loading, configuration, and unloading of your sound component:

- `SoundComponentInitOutputDevice`

- `SoundComponentSetSource`

- `SoundComponentGetSource`

- `SoundComponentGetSourceData`

- `SoundComponentSetOutput`

After the Sound Manager opens your sound component, it attempts to add your sound component to a sound component chain. Thereafter, the Sound Manager calls your component's `SoundComponentInitOutputDevice` function to give you an opportunity to set default values for any associated hardware and to perform any hardware-specific operations.

SoundComponentInitOutputDevice

A sound output device component must implement the `SoundComponentInitOutputDevice` function. The Sound Manager calls this function to allow a sound output device component to configure any associated hardware devices.

```
pascal ComponentResult SoundComponentInitOutputDevice
                            (ComponentInstance ti, long actions);
```

ti   A component instance that identifies your sound component.

actions   A set of flags. This parameter is currently unused.

DESCRIPTION

Your `SoundComponentInitOutputDevice` function is called by the Sound Manager at noninterrupt time to allow your sound output device component to perform any hardware-specific initialization. You should perform any necessary initialization that was not already performed in your `OpenComponent` function. Note that your `OpenComponent` function cannot assume that the appropriate hardware is available. As a result, the Sound Manager calls your `SoundComponentInitOutputDevice` function when it is safe to communicate with your audio hardware. You can call the `OpenMixerSoundComponent` function to create a single sound component chain.

SPECIAL CONSIDERATIONS

Your `SoundComponentInitOutputDevice` function is always called at noninterrupt time. All other component-defined routines might be called at interrupt time. Accordingly, your `SoundComponentInitOutputDevice` function should handle any remaining memory allocation needed by your component and it should lock down any relocatable blocks your component will access.

RESULT CODES

Your `SoundComponentInitOutputDevice` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See Listing 4-4 on page 282 for a sample `SoundComponentInitOutputDevice` function.

SoundComponentSetSource

A sound component can implement the `SoundComponentSetSource` function. The Sound Manager calls this function to identify your component's source component.

```
pascal ComponentResult SoundComponentSetSource
                            (ComponentInstance ti,
                                SoundSource sourceID,
                                ComponentInstance source);
```

ti   A component instance that identifies your sound component.

sourceID   A source ID for the source component chain created by the Apple Mixer.

source   A component instance that identifies your source component.

DESCRIPTION

Your `SoundComponentSetSource` function is called by the Sound Manager to identify to your sound component the sound component that is its source. The source component is identified by the `source` parameter. Your component uses that information when it needs to obtain more data from its source (usually, by calling its `SoundComponentGetSourceData` function).

Because a sound output device component is always connected directly to one or more instances of the Apple Mixer, the `SoundComponentSetSource` function needs to be implemented only by utility components (that is, components that perform modifications on sound data). Utility components are linked together into a chain of sound components, each link of which has only one input source. As a result, a utility component can usually ignore the `sourceID` parameter passed to it.

Your `SoundComponentSetSource` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentGetSource

A sound component can implement the `SoundComponentGetSource` function. The Sound Manager calls this function to determine your component's source component.

```
pascal ComponentResult SoundComponentGetSource
                        (ComponentInstance ti,
                            SoundSource sourceID,
                            ComponentInstance *source);
```

ti   A component instance that identifies your sound component.

sourceID   A source ID for the source component chain created by the Apple Mixer.

source   A component instance that identifies your source component.

DESCRIPTION

Your `SoundComponentGetSource` function is called by the Sound Manager to retrieve your component's source component instance. Your component should return, in the `source` parameter, the component instance of your component's source. This should be the source component instance your component was passed when the Sound Manager called your `SoundComponentSetSource` function.

In general, all sound components have sources, except for the source at the beginning of the source component chain. In the unlikely event that your component does not have a source, you should return `nil` in the `source` parameter. A sound output device component is always connected directly to an instance of the Apple Mixer. Accordingly, a sound output device component should return a component instance of the Apple Mixer in the `source` parameter and a source ID in the `sourceID` parameter. A utility component can ignore the `sourceID` parameter.

RESULT CODES

Your `SoundComponentGetSource` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentGetSourceData

A utility component must implement the `SoundComponentGetSourceData` function. A sound output device component calls this function on its source component when it needs more data.

```pascal
pascal ComponentResult SoundComponentGetSourceData
                            (ComponentInstance ti,
                                SoundComponentDataPtr *sourceData);
```

`ti`   A component instance that identifies your sound component.

`sourceData`
                On output, a pointer to a sound component data record that specifies the type and location of the data your component has processed.

DESCRIPTION

Your `SoundComponentGetSourceData` function is called when the sound component immediately following your sound component in the sound component chain needs more data. Your function should generate a new block of audio data, fill out a sound component data record describing the format and location of that data, and then return the address of that record in the `sourceData` parameter.

Your `SoundComponentGetSourceData` function might itself need to get more data from its source component. To do this, call through to the source component's `SoundComponentGetSourceData` function. If your component cannot generate any more data, it should set the `sampleCount` field of the sound component data record to 0 and return `noErr`.

IMPORTANT
Sound output device components do not need to implement this function, but all utility components must implement it. ▲

RESULT CODES

Your `SoundComponentGetSourceData` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentSetOutput

A sound output device component can call the `SoundComponentSetOutput` function of the Apple Mixer to indicate the type of data it expects to receive.

```
pascal ComponentResult SoundComponentSetOutput
                        (ComponentInstance ti,
                            SoundComponentDataPtr requested,
                            SoundComponentDataPtr *actual);
```

ti   A component instance that identifies your sound component.

requested
            A pointer to a sound component data record that specifies the type of the data your component expects to receive.

actual
            This parameter is currently unused.

DESCRIPTION

The Apple Mixer's `SoundComponentSetOutput` function can be called by a sound output device component to specify the kind of audio data the output device component wants to receive. The Apple Mixer uses that information to determine the type of sound component chain it needs to construct in order to deliver that kind of audio data to your sound output device component. For example, if your sound output device is able to accept 16-bit samples, the Sound Manager doesn't need to convert 16-bit audio data into 8-bit data.

The following lines of code illustrate how the sound output device component for the Apple Sound Chip might call Apple Mixer's `SoundComponentSetOutput` function:

```
myDataRec.flags = 0;                    /*ignored here*/
myDataRec.format = kOffsetBinary;   /*ASC needs offset binary*/
myDataRec.sampleRate = rate22khz;   /*ASC needs 22 kHz samples*/
myDataRec.sampleSize = 8;               /*ASC needs 8-bit data*/
myDataRec.numChannels = 2;              /*ASC can do stereo*/
myDataRec.sampleCount = 1024;       /*ASC uses a 1K FIFO*/
myErr = SoundComponentSetOutput(mySource, &myDataRec, &myActual);
```

In general, however, a sound output device component shouldn't need to call the Apple Mixer's `SoundComponentSetOutput` function. Instead, it can indicate the type of data it expects to receive when it calls the `OpenMixerSoundComponent` function. The `SoundComponentSetOutput` function is intended for sophisticated sound output device components that might want to reinitialize the Apple Mixer.

IMPORTANT
Only the Apple Mixer component needs to implement this function. ▲

RESULT CODES

The Apple Mixer's `SoundComponentSetOutput` function returns `noErr` if successful or an appropriate result code otherwise.

Creating and Removing Audio Sources

To write a sound output device component, you might need to define two routines that create and remove audio sources:

■ `SoundComponentAddSource`

■ `SoundComponentRemoveSource`

Your component needs to contain these functions only if, like the Apple Mixer, it can mix two or more audio channels into a single output stream. Sound components that operate on a single input stream only do not need to include these functions.

SoundComponentAddSource

A sound output device component that can mix multiple channel of audio data must implement the `SoundComponentAddSource` function to add a new sound source.

```
pascal ComponentResult SoundComponentAddSource
                    (ComponentInstance ti, SoundSource *sourceID);
```

ti   A component instance that identifies your sound component.
sourceID   On exit, a source ID for the newly created source component chain.

DESCRIPTION

The `SoundComponentAddSource` function is called by the Sound Manager to create a new sound source. If your sound output device component can mix multiple channels of sound, it needs to define this function. Your `SoundComponentAddSource` function should call the Sound Manager function `OpenMixerSoundComponent` to create an new instance of the Apple Mixer component. The Apple Mixer component then creates a sound component chain capable of generating the type of data your sound output device component wants to receive.

The Apple Mixer also assigns a unique 4-byte source ID that identifies the new sound source and component chain. You can retrieve that source ID by calling the Apple Mixer's `SoundComponentAddSource` function. Your `SoundComponentAddSource` function should then pass that source ID back to the Sound Manager in the `sourceID` parameter.

IMPORTANT

Most sound components do not need to implement the `SoundComponentAddSource` function. Only sound components that can handle more than one source of input need to define it. ▲

4

Sound Components

SPECIAL CONSIDERATIONS

The `SoundComponentAddSource` function is called at noninterrupt time.

RESULT CODES

Your `SoundComponentAddSource` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See page 298 for a description of `OpenMixerSoundComponent`.

SoundComponentRemoveSource

A sound output device component that implements the `SoundComponentAddSource` function must also implement the `SoundComponentRemoveSource` function to remove sound sources.

```
pascal ComponentResult SoundComponentRemoveSource
                    (ComponentInstance ti, SoundSource sourceID);
```

`ti`   A component instance that identifies your sound component.

`sourceID`   A source ID for the source component chain to be removed.

DESCRIPTION

Your `SoundComponentRemoveSource` function is called by the Sound Manager to remove the existing sound source specified by the `sourceID` parameter. Your `SoundComponentRemoveSource` function should do whatever is necessary to invalidate that source and then call through to the Apple Mixer's `SoundComponentRemoveSource` function.

IMPORTANT
Most sound components do not need to implement the `SoundComponentRemoveSource` function. Only sound components that can handle more than one source of input need to define it. ▲

SPECIAL CONSIDERATIONS

Your `SoundComponentRemoveSource` function is always called at noninterrupt time.

RESULT CODES

Your `SoundComponentRemoveSource` function should return `noErr` if successful or an appropriate result code otherwise.

Getting and Setting Sound Component Information

To write a sound component, you need to define two routines that determine the capabilities of your component or to change those capabilities:

- `SoundComponentGetInfo`
- `SoundComponentSetInfo`

SoundComponentGetInfo

A sound component must implement the `SoundComponentGetInfo` function. The Sound Manager calls this function to get information about the capabilities of your component.

```
pascal ComponentResult SoundComponentGetInfo
                            (ComponentInstance ti,
                                SoundSource sourceID,
                                OSType selector, void *infoPtr);
```

ti   A component instance that identifies your sound component.

sourceID   A source ID for a source component chain.

selector   A sound component information selector. See "Sound Component
            Information Selectors" beginning on page 287 for a description of the
            available selectors.

infoPtr   On output, a pointer to the information requested by the caller.

DESCRIPTION

Your `SoundComponentGetInfo` function returns information about your sound component. The `sourceID` parameter specifies the sound source to return information about, and the `selector` parameter specifies the kind of information to be returned. If the information occupies 4 or fewer bytes, it should be returned in the location pointed to by the `infoPtr` parameter. If the information is larger than 4 bytes, the `infoPtr` parameter is a pointer to a component information list, a 6-byte structure of type `SoundInfoList`:

```
typedef struct {
   short          count;
   Handle         handle;
} SoundInfoList, *SoundInfoListPtr;
```

This structure consists of a count and a handle to a variable-sized array. The `count` field specifies the number of elements in the array to which `handle` is a handle. It is your component's responsibility to allocate the block of data referenced by that handle, but it is the caller's responsibility to dispose of that handle once it is finished with it.

The data type of the array elements depends on the kind of information being returned. For example, the selector `siSampleSizeAvailable` indicates that you should return a list of the sample sizes your component can support. You return the information by passing back, in the `infoPtr` parameter, a pointer to an integer followed by a handle to an array of integers.

If your component cannot provide the information specified by the `selector` parameter, it should pass the selector to its source component.

SPECIAL CONSIDERATIONS

Your `SoundComponentGetInfo` function is not called at interrupt time if it is passed a selector that might cause it to allocate memory for the handle in the component information list.

RESULT CODES

Your `SoundComponentGetInfo` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See "Finding and Changing Component Capabilities" on page 283 for a sample `SoundComponentGetInfo` function.

SoundComponentSetInfo

A sound component must implement the `SoundComponentSetInfo` function. The Sound Manager calls this function to modify settings of your component.

```
pascal ComponentResult SoundComponentSetInfo
                            (ComponentInstance ti,
                                SoundSource sourceID,
                                OSType selector, void *infoPtr);
```

ti   A component instance that identifies your sound component.

sourceID   A source ID for a source component chain.

selector   A sound component information selector. See "Sound Component Information Selectors" beginning on page 287 for a description of the available selectors.

infoPtr   A pointer to the information your component is to use to modify its settings. If the information occupies 4 or fewer bytes, however, this parameter contains the information itself, not the address of the information.

DESCRIPTION

Your `SoundComponentSetInfo` function is called by the Sound Manager to set one of the settings for your component, as specified by the `selector` parameter. If the information associated with that selector occupies 4 or fewer bytes, it is passed on the stack, in the `infoPtr` parameter itself. Otherwise, the `infoPtr` parameter is a pointer to a structure of type `SoundInfoList`. See the description of `SoundComponentGetInfo` for more information about the `SoundInfoList` structure.

If your component cannot modify the settings specified by the `selector` parameter, it should pass the selector to its source component.

RESULT CODES

Your `SoundComponentSetInfo` function should return `noErr` if successful or an appropriate result code otherwise.

Managing Source Data

To write a sound output device component, you might need to define routines that manage the flow of data in a sound channel:

■ `SoundComponentStartSource`

■ `SoundComponentStopSource`

■ `SoundComponentPauseSource`

■ `SoundComponentPlaySourceBuffer`

SoundComponentStartSource

A sound output device component must implement the `SoundComponentStartSource` function. The Sound Manager calls this function to start playing sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentStartSource
                        (ComponentInstance ti,
                            short count, SoundSource *sources);
```

ti   A component instance that identifies your sound component.

count   The number of source IDs in the array pointed to by the `source` parameter.

sources   An array of source IDs.

DESCRIPTION

Your `SoundComponentStartSource` function is called by the Sound Manager to begin playing the sounds originating from the sound sources specified by the `sources` parameter. Your function should start (or resume) sending data from those sources to the associated sound output device. If your component supports only one sound source, you can ignore the `sources` parameter.

SPECIAL CONSIDERATIONS

Your `SoundComponentStartSource` function can be called at interrupt time.

RESULT CODES

Your `SoundComponentStartSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentStopSource

A sound output device component must implement the `SoundComponentStopSource` function. The Sound Manager calls this function to stop playing sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentStopSource
                            (ComponentInstance ti, short count,
                                SoundSource *sources);
```

ti   A component instance that identifies your sound component.
count   The number of source IDs in the array pointed to by the `source` parameter.
sources   An array of source IDs.

DESCRIPTION

Your `SoundComponentStopSource` function is called by the Sound Manager to stop the sounds originating from the sound sources specified by the `sources` parameter. Your function should stop sending data from those sources to the associated sound output device. In addition, your `SoundComponentStopSource` function should flush any data from the specified sound sources that it's caching. If your component supports only one sound source, you can ignore the `sources` parameter.

RESULT CODES

Your `SoundComponentStopSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentPauseSource

A sound output device component must implement the `SoundComponentPauseSource` function. The Sound Manager calls this function to stop pause the playing of sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentPauseSource
                            (ComponentInstance ti,
                                short count, SoundSource *sources);
```

ti   A component instance that identifies your sound component.

count   The number of source IDs in the array pointed to by the `source` parameter.

sources   An array of source IDs.

DESCRIPTION

Your `SoundComponentPauseSource` function is called by the Sound Manager to pause the playing of the sounds originating from the sound sources specified by the `sources` parameter. Your function should stop sending data from those sources to the associated sound output device. Because your `SoundComponentStartSource` function might be called to resume playing sounds, you should not flush any data. If your component supports only one sound source, you can ignore the `sources` parameter.

RESULT CODES

Your `SoundComponentPauseSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentPlaySourceBuffer

A sound component must implement the `SoundComponentPlaySourceBuffer` function. The Sound Manager calls this function to start a new sound playing.

```
pascal ComponentResult SoundComponentPlaySourceBuffer
                            (ComponentInstance ti,
                                SoundSource sourceID,
                                SoundParamBlockPtr pb,
                                long actions);
```

ti   A component instance that identifies your sound component.

sourceID   A source ID for a source component chain.

pb   A pointer to a sound parameter block.

actions   A set of 32 bit flags that describe the actions to be taken when preparing to play the source data. See "Action Flags" on page 292 for a description of the constants you can use to select bits in this parameter.

DESCRIPTION

Your `SoundComponentPlaySourceBuffer` function is called by the Sound Manager to start a new sound playing. The sound parameter block pointed to by the `pb` parameter specifies the sound to be played. That parameter block should be passed successively to all sound components in the chain specified by the `sourceID` parameter. This allows the components to determine their output formats and playback settings and to prepare for a subsequent call to their `SoundComponentGetSourceData` function. It also allows a sound output device component to prepare for starting up its associated hardware.

RESULT CODES

Your `SoundComponentPlaySourceBuffer` function should return `noErr` if successful or an appropriate result code otherwise.

Sound Manager 3.3 Features

This chapter discusses new features of the Sound Manager that are available since the release of Sound Manager 3.0. The Sound Manager is described in Chapter 2, "Sound Manager." You need to read this chapter if you are writing an application and want to add sound to it.

This chapter expands on the release notes that accompany the Sound Manager 3.3 software. For example, Sound Manager 3.3 now supports scheduling of sounds, both in the past and in the future. In addition, Sound Manager sources are now multi-platform, which means that the same manager is available for each platform, i.e., Mac OS, Windows 95 and NT, and contains the same API and features.

Notably, in Sound Manager 3.3, the contents of both SoundInput.h and SoundComponents.h interface files have been merged into a single Sound.h file.

In addition to documenting the changes in the current release, this chapter also discusses some of the features available in Sound Manager 3.2.1 and 3.1, including two new audio codecs, performance enhancements, and asynchronous alert sounds.

Read this chapter if you want to use the Sound Manager to

- determine the currently installed version

- convert between sound formats using the `SoundConverterOpen` function

- use the sound conversion architecture to convert a buffer of silence to IMA 4:1, changing the sampling rate in the process.

These are described in the section "Using the Sound Manager" (page 323).

Chapter 6, "Sound Manager 3.3 Reference," explains the new sound commands, sound informational selectors and Sound Manager routines added since the release of Sound Manager 3.0 (as documented in the rest of this book).

Features of Sound Manager 3.2.1

This section describes some of the features that are only available with Sound Manager version 3.2.1 or later. Check for this by calling the `SndSoundManagerVersion` routine for the installed version.

Sound Manager 3.1 added two new audio codecs, significant performance increases on the Power Macintosh line of computers, and asynchronous alert sounds. It is completely backwards compatible with previous versions of the Sound Manager.

Pre-mixer Effects

The `siPreMixerSoundComponent` has been added for installing a pre-mixer sound component to provide specialized audio effects, such as the Sound Sprockets sound localizer. Refer to the new Sprockets Games documentation for use with this new sound effect.

Native PowerPC Code

All of the critical sound components are native PowerPC code. This includes the mixer, sample rate converter, format converters, MACE 3:1 and 6:1 compression and decompressers. The sound input interrupt handler also uses native PowerPC code to boost performance during recording. These enhancements on Power Macintosh will let games and QuickTime movies play more smoothly and provide higher capture rates when recording digital video.

Three New Audio Codecs Added

The IMA 4:1 audio compression format is based on a standard proposed by the Interactive Multimedia Association, and is used to compress 16-bit sound with a ratio of 4:1. It is particularly good at compressing CD-quality music and is fully integrated into QuickTime.

The μLaw 2:1 format (pronounced "mu-law") is an international standard for compressing voice-quality audio (typically 16-bit, 8 kHz speech) with a ratio of

2:1. It is often used in telephony applications, and also on the Internet as the encoding format for ".au" sound files.

The 16-bit "little-endian" codec provides support for the byte-ordering used on Intel-standard personal computers. This codec lets QuickTime directly play 16-bit sound files encoded in the popular ".wav" file format.

Playing Alert Sounds Asynchronously

Previous versions of the Sound Manager would hang your Macintosh computer while playing an alert sound, forcing you to wait until the sound was done playing before you could continue. Sound Manager 3.1 removed this limitation by playing alert sounds asynchronously, so alert dialogs and other interface elements can continue processing while the alert sound is playing.

New sounds are queued within the same process and additional sounds from other processes are mixed together. This gives the user a perceived performance increase. You'll probably feel that the machine is running faster when a system alert sound doesn't lock up your computer anymore. You can disable this by setting the `sysBeepSynchronous` flag when calling `SndSetSysBeepState`.

New Features of Sound Manager 3.3

The new features of the Sound Manager 3.3 are described in the following section.

General

- Better optimization of PowerPC code.

- Unlimited output sources in mixer now available. This means you can open as many sound channels as memory will allow. The CPU load is determined by which ones are actively playing sound. If the load is beyond the capabilities of the computer, the sound will glitch or have gaps in it.

- Sound, `'snd '`, resources defined to be native endian format. Calling GetResource(soundListRsrc, 1) under QuickTime will perform endian swapping of all values in the resource. The Sound Manager API assumes any `'snd '` used (for example, the `SndPlay` function) will be in the native endian

format. The `SetupSndHeader` function will create a header in native endian format as well.

■ The sample rate converter does not round sloppy rates to the nearest real rate.

■ Sound Manager built-in sound codecs support the `siCompressionFactor` selector for the functions `SndSetInfo` and `SndGetInfo`. This allows the codec to support different settings, such as a different frame size.

■ The Sound Manager issues the `siCompressionFactor` selector on the sound channel before calling the `GetCompressionInfo` routine. The `GetCompressionInfo` is a generic call to a codec that is not attached to a channel. Using `siCompressionFactor` allows a channel to have compression or decompression settings which are specific to that channel's decompressor.

The Ability to Schedule Sounds

A major new feature of Sound Manager 3.3 is the ability to schedule a sound. The mixer supports scheduling sounds, both in the past and future. If the start time is in the past, the mixer will fast forward through the source at non-interrupt time, attempting to catch up with the current time. Once there, the sounds are included into the mix. If the sound is in the future or the past, the mixer will begin mixing the source into the output stream at the specified time within a sample of accuracy.

New Sound Commands

There are two new sound commands that support the sound clock: `clockComponentCmd` and `getClockComponentCmd`. These commands are used by QuickTime to obtain the sound clock which is used as the main time base. For more information about these new sound commands, see Chapter 6, "Sound Manager 3.3 Reference."

The Sound Manager sound clock is only available when QuickTime is installed. This allows QuickTime's video to be synchronized with the audio stream, and can be used to synchronize multiple audio sources as well. There is one sound clock per source (for example, sound channel) which is updated with each hardware buffer. The sound clock counts samples being consumed by the hardware device as it is passed from the Mixer.

Once the clock ComponentInstance has been obtained, any and all of the QuickTime clock component calls (for example, `ClockGetTime`) can be used.

Multi-platform Support

The Sound Manager sources are now multi-platform, i.e., built for the Mac OS, Windows 95 and NT. This means the same Sound Manager is available for each platform, and contains the same API and features.

The Issue of "Endianness"

Multi-platform support raises the issue of "endianness." Basically, endian conversion is treated exactly as a compression conversion. Any non-native endian format will be required to be "decompressed" into the native format. There are three distinct groups to be considered here:

■ a program playing audio

■ a sound component manipulating the audio

■ the output device sending the audio to a hardware device

For applications playing audio for any non-native endian formats, you use the `CmpSoundHeader`, `SndDoubleBufferHeader2`, or `SoundComponentData` with the format set to `k16BitLittleEndianFormat`. This causes the Sound Manager to install the 16 Bit Little Endian codec into the sound channel. (Note that the `SoundHeader`, `ExtSoundHeader`, and `SndDoubleBufferHeader` structures do not have a format field.)

Note
If you don't want to figure whether it's native or not, just alert use of these headers and fill in the format field. The Sound Manager will do the right thing. ◆

For sound component developers, there are two cases to consider: compressor and decompressor components. The uncompressed formats are assumed to be in native format of the platform. In other words, the input to a compressor is in native format and all output from a decompressor is in native format.

For sound output devices, the Mixer is generating audio in the platform's native format. If the hardware requires non-native format, it is the responsibility of the output device to convert from the native format.

Sound Formats

The new extended precision formats (24- and 32-bit integers and 32- and 64-bit floats) supported by Apple Computer will also support both big and little endian format. The Apple sound components will assume big endian format, unless otherwise configured using the `siDecompressionParams` or `siCompressionParams` selector. Decompression components should always output native format, and default to big endian for their input. Compression components should default to big endian format for output, and consume native endian for input.

A New Sound Codec Included: a-Law Compression

A new sound codec has been included with the standard set of built-in compression format: the aLaw compression. The µLaw format is used in North America and Japan, while the aLaw format is used in Europe and the rest of the world.

New Sample Sizes Supported: 24 and 32 Bit Integer

Two new sample sizes are supported, integer 24 and 32 bit, and named `k24BitFormat` and `k32BitFormat`, respectively. For this release, these samples are converted into 8 or 16 bit depending on the hardware.

New Codecs to Import & Export Samples

Two new codecs have been added to import and export samples. These are in 32 bit and 64 bit floating point formats, and named `kFloat32Format` and `kFloat64Format`, respectively. This allows more accurate conversion to other sample sizes or the application of effects—for example, 3D sound localization. These two floating point codecs also support a new selector `siSlopeAndIntercept` which points to a `SoundSlopeAndInterceptRecord` structure which can control the conversion process of floating point samples into integer samples. This structure is included in the Sound.h interface file and shown as follows:

```
struct SoundSlopeAndInterceptRecord {
    Float64   slope;
    Float64   intercept;
```

```
    Float64   minClip;
    Float64   maxClip;
};
```

New Functions Added to the SoundConverter Suite

Two new function calls have been added to the `SoundConverter` suite:
`SoundConverterGetInfo` and `SoundConverterSetInfo`. This allows any of the
sound info selectors and their parameters to be used when calling the
`SoundConverter`.

New Sound Info Selectors Created

Two new sound info selectors have been created: `siDecompressionParams` and
`siCompressionParams`. These selectors are used to pass an atom list to the sound
component. Atoms are always in big-endian format. The content of the new
audio atom list can consist of many atoms, and always ends with the
AudioTerminatorAtom. The first atom in the list should be the
AudioFormatAtom, which specifies which sound component is responsible for
the atoms contained within the list.

If your application calls the `SndGetInfo` function with these two selectors, it will
return a handle to an atom list. This handle is the responsibility of the caller to
be disposed of. The `SndSetInfo` function requres a pointer to an atom list, and is
the responsiblity of the caller to be disposed. For example, to change a
decompression sound component from its default of big endian to little endian
input, you use the AudioEndianAtom with the "littleEndian" field set to `TRUE`.
Listing 1 shows an example of changing the 32- bit floating point codec to
consume little endian format.

Listing 5-1   Changing the 32 bit floating point codec to consume little endian format

```
void *atoms;

err = CreateAudioAtomsList(&atoms, k32BitFormat, false);
if (err != noErr) return (err);

err = SndSetInfo(chan, siDecompressionParams, atoms);
```

Using the siOptionsDialog Selector

Certain sound compression components can show a dialog of options. To determine if the codec supports a dialog, use the SoundComponentGetInfo function with the siOptionsDialog selector. It will return a 16 bit value, with any non-zero value meaning TRUE.

If you use the SoundComponentSetInfo function, the codec will show the dialog and change its settings accordingly, as shown in Listing 2. For example, the two floating point and the two new 24 and 32 bit integer codecs can show a dialog that allows the user to specify big or little endian. You can also change a compressor's setting using the siCompressionParams selector.

Listing 5-2   Using the SoundComponentSetInfo function to show the dailog of options

```
short hasDialog;

err = SoundComponentGetInfo(compressor, nil, siOptionsDialog,
&hasDialog);
if (err != noErr) return (err);

if (hasDialog)
{
    err = SoundComponentSetInfo(compressor, nil, siOptionsDialog, nil);
    if (err != noErr) return (err);
}
```

Interface Changes

In Sound Manager 3.3, SoundInput.h and SoundComponents.h have been merged into Sound.h. The contents of the two former interface files are now empty; all sound interfaces have been moved to Sound.h.

Previously defined in SoundComponents.h were a set of component sub-types used in the 'thng' resource of an audio codec. This OSType was matched with a given sound's format, which determined the codec necessary to support the given sound.

Unfortunately, the usage of these constants was not obvious to many and are required as the value set in the format parameter of the functions

`GetCompressionInfo`, `GetCompressionName`, `SetupSndHeader`, **and** `SetupAIFFHeader`, **and in the** `CmpSoundHeader`, `SndDoubleBufferHeader2`, `CompressionInfo` **and** `SoundComponentData` structures. A new set of constants has been created to replace these, along with the addition of the recently supported formats. These are shown in Table 1.

Table 5-1   A new set of constants, replacing old constants

New Constants	Old Constants
kSoundNotCompressed	
k8BitOffsetBinaryFormat	kOffsetBinary
kMACE3Compression	kMace3SubType
kMACE6Compression	kMace6SubType
kIMACompression	kIMA4SubType
kULawCompression	kULawSubType
kALawCompression	
kFloat32Format	
kFloat64Format	
k24BitFormat	
k32BitFormat	
k16BitBigEndianFormat	kTwosComplement
k16BitLittleEndianFormat	kLittleEndianSubType
kMicrosoftADPCMFormat	
kDVIIntelIMAFormat	
kDVAudioFormat	

Using the Sound Manager

The following section discusses some of the ways that you can use the Sound Manager.

Determining the Sound Manager Version

The Sound Manager provides a routine, `SndSoundManagerVersion`, which returns the currently installed version.

Listing 3 shows the simplest method you can use to check for Sound Manager 3.1 or later. Note that the `NumVersionVariant` structure is being used from the lastest version of the MacTypes interface file. This is a union of the old `NumVersion` structure with its various parts, and a 32 bit long value which allows for easy comparisons.

Listing 5-3 The simplest method for determing the Sound Manager version

```
Boolean HasSoundManager3_1(void)
{
NumVersionVariant    version;

version.parts = SndSoundManagerVersion();
return (version.whole >= 0x03100000)            // version 3.1
}
```

Converting Between Sound Formats

With the release of Sound Manager 3.2, you can easily convert between sound formats. Typically, some of the operations that can be performed include

■ compression

■ decompression

■ channel conversion

■ sample rate conversion

■ sample format conversion

You begin a conversion session by calling the `SoundConverterOpen` function, to which you pass the format of the sound to be converted and the desired output format. A SoundConverter identifier is returned that must be passed to all further routines in this session. `SoundConverterClose` is used to close the session.

The `SoundConverterGetBufferSizes` routine allows you to determine input and output buffer sizes based on a target buffer size. This lets you allocate buffers to fit the conversion established with `SoundConverterOpen`.

The Process of Converting a Sound

The process of Converting a sound involves the following steps:

1. You call `SoundConverterBeginConversion` to initiate the conversion and reset the SoundConverter to default settings.

2. `SoundConverterConvertBuffer` is called one or more times to convert sequential buffers of the input data to the output format.

3. Finally, when all input data has been converted, `SoundConverterEndConversion` flushes out any data left in the converter.

```
pascal OSErr SoundConverterOpen(const SoundComponentData *inputFormat,
            const SoundComponentData *outputFormat, SoundConverter *sc)
```

`SoundConverterOpen` sets up the conversion session and returns a `SoundConverter` identifier to be passed to all further routines. The `inputFormat` parameter specifies the format of the sound data to be converted using a `SoundComponentData` structure. The following fields must be set up to describe the sound correctly:

Field Descriptions of SoundComponentData

`flags`	Set to 0
`format`	The sound format (i.e., 'raw', 'twos', 'MAC3', etc.)
`numChannels`	The number of channels (i.e., 1 = mono, 2 = stereo)
`sampleSize`	The sample size (i.e., 8 = 8-bit, 16 = 16-bit)
`sampleRate`	The sampling rate (in samples/second)
`sampleCount`	Set to 0
`buffer`	Set to 0
`reserved`	Set to 0

The `outputFormat` parameter specifies the output format, and must be passed fields similar to inputFormat. Output fields that are different from input fields will cause a conversion. For example, if the input sound format is `'raw '` and the output format is `'MAC3'`, the data resulting from the conversion will be compressed with MACE 3:1. This allows any combination of compression,

decompression, channel conversion, sample size conversion and sampling rate conversion. A `SoundConverter` identifier is returned to manage the session, which must be passed to all further routines.

`SoundConverterClose` terminates the session and frees up all memory and services associated with this session.

```
pascal OSErr SoundConverterClose(SoundConverter sc)
```

An Example of Converting a Buffer of Silence to IMA 4:1

The following is an example of how to use the sound conversion architecture to convert a buffer of silence to IMA 4:1, changing the sampling rate in the process.

```
enum {
    kTargetBytes = 20 * 1024
};

void main(void)
{
    SoundConverter sc;
    SoundComponentData inputFormat, outputFormat;
    unsigned long inputFrames, inputBytes;
    unsigned long outputFrames, outputBytes;
    Ptr inputPtr, outputPtr;
    OSErr err;

    inputFormat.flags = 0;
    inputFormat.format = kOffsetBinary;
    inputFormat.numChannels = 1;
    inputFormat.sampleSize = 8;
    inputFormat.sampleRate = rate22050hz;
    inputFormat.sampleCount = 0;
    inputFormat.buffer = nil;
    inputFormat.reserved = 0;

    outputFormat.flags = 0;
    outputFormat.format = kIMA4SubType;
    outputFormat.numChannels = 1;
    outputFormat.sampleSize = 16;
```

```
    outputFormat.sampleRate = rate44100hz;
    outputFormat.sampleCount = 0;
    outputFormat.buffer = nil;
    outputFormat.reserved = 0;

    err = SoundConverterOpen(&inputFormat, &outputFormat, &sc);
    if (err != noErr)
        DebugStr("\pOpen failed");

    err = SoundConverterGetBufferSizes(sc, kTargetBytes,
                    &inputFrames, &inputBytes, &outputBytes);
    if (err != noErr)
        DebugStr("\pGetBufferSizes failed");

inputPtr = NewPtrClear(inputBytes);
outputPtr = NewPtrClear(outputBytes);

// fill input buffer with 8-bit silence
{
    int i;
    Ptr dp = inputPtr;

    for (i = 0; i < inputBytes; i++)
        *dp++ = 0x80;
}

    err = SoundConverterBeginConversion(sc);
    if (err != noErr)
        DebugStr("\pBegin Conversion failed");

    err = SoundConverterConvertBuffer(sc, inputPtr, inputFrames,
                    outputPtr, &outputFrames, &outputBytes);
    if (err != noErr)
        DebugStr("\pConversion failed");

    err = SoundConverterEndConversion(sc,
                outputPtr,&outputFrames, &outputBytes);
    if (err != noErr)
        DebugStr("\pEnd Conversion failed");
```

```
    err = SoundConverterClose(sc);
    if (err != noErr)
        DebugStr("\pClose failed");
}
```

Scheduling Two Sounds with the Sound Clock

Listing 4 shows an example of scheduling two sounds with the sound clock. It assumes that the SoundHeader of the ScheduledSoundHeader structure has already been established. Note that although each channel has its own clock, both are synchronized with the Mixer. Therefore, obtaining the current time from one channel will result in the same time for the second. This allows both channels to use the same start time.

Listing 5-4   An example of scheduling two sounds with the sound clock

```
// turn on the sound clock for two channels

cmd.cmd = clockComponentCmd;
cmd.param1 = true;
err = SndDoImmediate(gChan1, &cmd);
if (err != noErr) return (err);

err = SndDoImmediate(gChan2, &cmd);
if (err != noErr) return (err);

// get the sound clock component from one channel

cmd.cmd = getClockComponentCmd;
cmd.param2 = (long)&clock;
err = SndDoImmediate(gChan1, &cmd);
if (err != noErr) return (err);
// get the current time from one channel

scheduledSound1.startTime.base = nil;
err = ClockGetTime(clock, &scheduledSound1.startTime);
if (err != noErr) return (err);

// add 1/2 second to the current time
```

```
deltaTime.value.hi = 0;
deltaTime.value.lo = 1;
deltaTime.scale = 2;
AddTime(&scheduledSound1.startTime, &deltaTime);

// schedule two sounds to play at the same starting time

scheduledSound1.flags = kScheduledSoundDoScheduled;
scheduledSound2.flags = kScheduledSoundDoScheduled;
scheduledSound2.startTime = scheduledSound1.startTime;

cmd.cmd = scheduledSoundCmd;
cmd.param2 = (long)&scheduledSound1;
err = SndDoImmediate(gChan1, &cmd);
if (err != noErr) return (err);

cmd.param2 = (long)&scheduledSound2;
err = SndDoImmediate(gChan2, &cmd);
if (err != noErr) return (err);
```

Note

A sound channel can only have one scheduled sound at any given time. The Mixer does not queue up additional scheduled sounds. You can stream a sound continuously by starting the first buffer with the `scheduledSoundCmd`, then use the `bufferCmd` and `callBackCmd` sequence with the `SndDoCommand` function to stream additional samples. This has been the techniqued used in the past. By replacing the first `bufferCmd` with the new `scheduledSoundCmd`, you can cause the samples to start playing at a specific point in time. ◆

Converting to 32-bit Little Endian Data

A code example for converting to 32 bit little endian data is shown in Listing 5.

Listing **5-5**   Converting to Little Endian

```
OSErr ConvertToLittleEndian(Ptr inputPtr, Ptr outputPtr)
{
    SoundConverter        sc;
    SoundComponentData    inputFormat, outputFormat;
    unsigned long         inputFrames, inputBytes;
    unsigned long         outputFrames, outputBytes;
    void                  *littleEndianAtomsList;
    OSErr                 err;

    inputFormat.flags = 0;
    inputFormat.format = k16BitBigEndianFormat;
    inputFormat.numChannels = 1;
    inputFormat.sampleSize = 16;
    inputFormat.sampleRate = rate22050hz;
    inputFormat.sampleCount = 0;
    inputFormat.buffer = nil;
    inputFormat.reserved = 0;

    outputFormat.flags = 0;
    outputFormat.format = k32BitFormat;
    outputFormat.numChannels = 1;
    outputFormat.sampleSize = 16;
    outputFormat.sampleRate = rate22050hz;
    outputFormat.sampleCount = 0;
    outputFormat.buffer = nil;
    outputFormat.reserved = 0;

    err = SoundConverterOpen(&inputFormat, &outputFormat, &sc);
    if (err != noErr) return (err);

    err = SoundConverterGetBufferSizes(sc, kOurInputBytesTarget,
                        &inputFrames,&inputBytes, &outputBytes);
    if (err != noErr) return (err);

    err = CreateAudioAtomsList(k32BitFormat, true,
                             &littleEndianAtomsList);
    if (err != noErr) return (err);
```

```
    err = SoundConverterSetInfo(sc, siCompressionParams,
                                littleEndianAtomsList);
    if (err != noErr) return (err);

    err = SoundConverterBeginConversion(sc);
    if (err != noErr) return (err);

    err = SoundConverterConvertBuffer(sc, inputPtr, inputFrames,
                                outputPtr,&outputFrames, &outputBytes);
if (err != noErr) return (err);

    err = SoundConverterEndConversion(sc, outputPtr, &outputFrames,
                                    &outputBytes);
    if (err != noErr) return (err);

    err = SoundConverterClose(sc);
    if (err != noErr) return (err);
}
```

A List of Audio Atoms

This section is applicable to codec component developers.

Audio atoms are aways in big endian format. The atoms contained in this list can be in any order, ending with the AudioTerminatorAtom. No assumptions should be made about reading an audio atom list, other than the last atom is the AudioTerminatorAtom. Using the SndGetInfo function and the siDecompressionParams selector will return a list of atoms of any possible ordering, as shown in Listing 5-6.

Listing 5-6   A list of audio atoms of any possible ordering

```
OSErr CreateAudioAtomsList(OSType format, Boolean littleEndian,
                void **littleEndianAtomsList)
{
typedef struct {
AudioFormatAtom        formatData;
AudioEndianAtom        endianData;
AudioTerminatorAtom    terminatorData;
} AudioDecompressionAtoms, *AudioDecompressionAtomsPtr;
```

```
atoms =
(AudioDecompressionAtomsPtr)NewPtr(sizeof(AudioDecompressionAtoms));
if (atoms == nil)
    return (MemError());

atoms->formatData.size = EndianU32_NtoB(sizeof(AudioFormatAtom));
atoms->formatData.atomType = EndianU32_NtoB(kAudioFormatAtomType);
atoms->formatData.format = EndianU32_NtoB(format);

atoms->endianData.size = EndianU32_NtoB(sizeof(AudioEndianAtom));
atoms->endianData.atomType = EndianU32_NtoB(kAudioEndianAtomType);
atoms->endianData.littleEndian = EndianU16_NtoB(littleEndian);

atoms->terminatorData.size = EndianU32_NtoB(sizeof(AudioTerminatorAtom));
atoms->terminatorData.atomType =
EndianU32_NtoB(kAudioTerminatorAtomType);

*littleEndianAtomsList = atoms;
return (noErr);
}
```

Listing 5-7   How to read a list of audio atoms

```
Boolean GetFormatAndEndianFromAtomsList(UserDataAtom *atom, OSType
                                        *format, Boolean *littleEndian)
{
    Boolean  moreAtoms;

    moreAtoms = true;
    do
    {
        if (EndianS32_BtoN(atom->size) < 8)
        return (false);// bad atom size
    switch (EndianU32_BtoN(atom->atomType))
    {
    case kAudioFormatAtomType:
        *format = EndianU32_BtoN(((AudioFormatAtom *)atom)->format);
        break;
```

```
    case kAudioEndianAtomType:
        *littleEndian = EndianU16_BtoN(((AudioEndianAtom
                                        *)atom)->littleEndian);
        break;

    case kAudioTerminatorAtomType:
        moreAtoms = false;
        break;


    default:                            // unknown atom type
        break;
}
atom = (UserDataAtom *)((long)atom + EndianS32_BtoN(atom->size));
    } while (moreAtoms);
}
```

Using the SoundLib Shared library (PowerPC Only)

If you are developing a PowerPC-native application and wish to call some of the new routines in Sound Manager 3.0 and later, you need to link to the SoundLib shared library. This is because not all of the Sound Manager routine definitions are included in the InterfaceLib library currently built into Power Macintosh systems. The Sound Manager 3.1 extension installs a shared library with these missing routine definitions, and you use SoundLib to reference this library.

SoundLib is a "dummy" library, i.e., it contains just symbol references to the real shared library in the Sound Manager 3.1 extension. The SoundLib file should be used only for linking your PowerPC application—it should *not* be installed in the System Folder and is not to be given to users, because this can cause library conflicts. The SoundLib file simply provides the symbols that are then resolved from the Sound Manager 3.1 extension at run time. You should "weak" link with SoundLib and check in your application for the presence of Sound Manager 3.1 before calling one of these new routines (see sample code above). SoundLib is a .pef file, not a .xcoff file.

Some of the routines defined in SoundLib include `GetCompressionInfo`, `GetSoundPreference`, `SetSoundPreference`, `UnsignedFixedMulDiv`, `SndGetInfo`, `SndSetInfo`, and all the sound component interfaces needed when developing a native sound component.

Sound Manager 3.3 Reference

This chapter discusses the new sound commands, sound informational selectors, and Sound Manager routines added since the release of Sound Manager 3.0 as documented in the rest of this book.

New API Elements

The following are new sound commands, sound informational selectors, and Sound Manager routines added since the release of Sound Manager 3.0, as documented in the rest of this book.

Sound Commands

New sound commands have been added since the release of Sound Manager 3.0.

Use these sound commands with the `SndDoImmediate` and `SndDoCommand` functions.

clockComponentCmd

This command turns the sound clock on or off. Use this command to turn on the sound clock before attempting to use the `getClockComponentCmd`. Set the value of `param1` in this command to be `true` or `false`, where `true` turns on the clock.

```
SndCommand       cmd;

cmd.cmd = clockComponentCmd;
cmd.param1 = true; // turn the clock on
err = SndDoImmediate(chan, &clockComponentCmd);
```

getClockComponentCmd

This command returns the the sound clock component instance. Set the value of param2 to be a pointer to a ComponentInstance.

```
SndCommand          cmd;
ComponentInstance   clock;

cmd.cmd = getClockComponentCmd;
cmd.param2 = (long)&clock; // get the clock component
err = SndDoImmediate (chan, &getClockComponentCmd);
```

scheduledSoundCmd

Use this command with the new ScheduledSoundHeader structure. It is similar to the bufferCmd in that param2 contains a pointer to this structure. This command can be used to schedule a sound and/or to install a callBackCmd with this one command. The first field of this structure is a union of one of the three existing SoundHeader types, as used in the bufferCmd. The flags field is used to specify if there is a valid TimeRecord and/or parameters for the CallBackProc.

```
/* ScheduledSoundHeader flags*/
enum {
    kScheduledSoundDoScheduled = 1 << 0,
    kScheduledSoundDoCallBack = 1 << 1
};

struct ScheduledSoundHeader {
    SoundHeaderUnion   u;
    long        flags;
```

```
    short       reserved;
    short       callBackParam1;
    long        callBackParam2;
    TimeRecord  startTime;
};
```

linkSoundComponentsCmd

Use this command to configure the given sound channel's components to support your sound format. By default, a channel will be configured to play 8 bit non-compressed samples. If you wanted to play 16 bit or compressed audio, then you had to start your sound at non-interrupt time so that the proper sound components would be opened and installed. By using this new command you can point to a `SoundComponentData` structure in `param2`, and the proper set of sound components will be configured into your channel. At this point you can start a sound playing at interrupt time.

rateMultiplierCmd

The `rateMultiplierCmd` uses a fixed-point value to provide a multiplier to the playback rate of all sounds played on this channel. This allows you to vary the sample rate of the sound being played, and thus control its pitch. The `getRateMultiplierCmd` returns the current rate multiplier. For example, to play all sounds on a channel shifted up one octave in pitch, you could use the following code:

```
                OSErr RaisePitchOneOctave(SndChannelPtr chan)
                {
                        SndCommand cmd;
                        OSErr err;

                        cmd.cmd = rateMultiplierCmd;
                        cmd.param1 = 0;
                        cmd.param2 = 0x00020000; // rate of 2.0
```

```
                             err = SndDoImmediate(chan, &cmd);
                             return (err);
   }
```

The `rateMultiplierCmd` is more useful than previous
versions of `rateCmd`, which applied only to the sound
currently playing and was based on the sampling rate of
the hardware.

Sound Informational Selectors

The following are new sound informational selectors added since the release of
Sound Manager 3.0. These selectors are used to obtain information and control
the sound environment using the `GetSoundOutputInfo`, `SetSoundOutputInfo`,
`SndGetInfo`, `SndSetInfo`, or `SPBGetDeviceInfo`, and `SPBSetDeviceInfo` routines.

siHardwareBusy

This selector is used by all input and output devices, and returns the state of the
hardware—typically, whether or not hardware interrupts are active. For input
devices, the `infoData` parameter points to a short word. For output devices, the
`infoPtr` is a short word containing the value.

A value of 0 represents that the hardware is not busy, while a value of 1
represents busy. This selector is only supported by the get calls, since it does not
make sense to "set" a device busy.

```
short hwBusy;
err = GetSoundOutputInfo(nil, siHardwareBusy, &hwBusy);
```

siHardwareFormat

The `siHardwareFormat` selector has been added for output devices. Use this
selector with the `GetSoundOutputInfo` and `SndGetInfo` routines. It returns a
`SoundComponentData` structure of the format used by the output hardware device.

```
SoundComponentData outputFormat;

err = GetSoundOutputInfo(nil, siHardwareFormat, &outputFormat);
```

siHardwareMute

This output selector was previously defined, but should be documented and implemented as returning the mute state of all sources that can be heard. For example, Power Macintosh computers have a speaker and headphone source. If no headphones are plugged in, then return the mute state of the speakers. If headphones are inserted, then return the mute state of both (e.g. `siHardwareMute` is muted when both speakers and headphones are muted). The idea is if the user has sources that can be heard and all are muted, then the hardware has been muted. If any one of the sources can be heard, then the hardware is not muted.

```
short hwMuted;

err = GetSoundOutputInfo(nil, siHardwareMute, &hwMuted);
```

siHardwareVolume

This output selector was previously defined, but should be documented and implemented as returning the volume of the device that can be heard. For example, Power Macintosh computers can have the speaker and the headphones both be audible to the user.

In this case, return the loudest volume setting. If both the speaker and the headphones are muted, then again return the loudest setting. If one is muted and the other is audible, then return the volume of the audible source.

```
unsigned long hwVolume;

err = GetSoundOutputInfo(nil, siHardwareVolume, &hwVolume);
```

siPreMixerSoundComponent

This selector is used to install a given sound component into the component chain. The given component will be installed just before the mixer component. This sound component can access the audio stream and modify the data prior to its being sent to the mixer. An example of its use is shown as follows by installing the Sound Sprockets sound localizer.

```
SoundComponentLink soundLink;

soundLink.description.componentType = kSoundEffectsType;
soundLink.description.componentSubType = kSSpLocalizationSubType;
soundLink.description.componentManufacturer = kAnyComponentManufacturer;
soundLink.description.componentFlags = 0;
soundLink.description.componentFlagsMask = kAnyComponentFlagsMask;
soundLink.mixerID = nil;
soundLink.linkID = nil;

err = SndSetInfo(chan, siPreMixerSoundComponent, &soundLink);
// configure the localization settings
// and send them to the sound component

err = SndSetInfo(chan, siSSpLocalization, &localizationSettings);
```

siSetupCDAudio

This selector is only used by input devices that have special requirements in order to hear the audio from the CD player. For example, currently the 840AV and Power Macintosh computers require the user to open the Sound control panel, select the Inputs panel, and then open the Options dialog. From here they have to select the CD Input and also check the Play-Through option. The new `siSetupCDAudio` avoids this troublesome operation by setting up the input device to allow you to hear audio CDs from the audio CD player.

The `infoData` parameter points to a short word. A value of 1 means to set up the hardware (for example, set input source to CD and turn on Play-Through). A value of 0 means to return the input hardware to some initial state, either the

default settings or the settings prior to setting up for the CD audio. Any other value is an error.

If the input device had no special needs for CD audio (for exmaple, the audio is heard regardless of the input hardware settings), then the selector is not supported and it returns an error. The SPBGetDeviceInfo routine should return a 1 or 0 value, depending on its current setting. If the input source is the CD and Play-Through, if required, is on, then return the value of 1. This should happen even if nothing ever requested setting up for CD audio. It should return the state of the actual condition of the input hardware.

Sound Manager Routines

The following are new Sound Manager routines added since the release of Sound Manager 3.0.

SndGetInfo and SndSetInfo

```
extern pascal OSErr SndGetInfo(
                    SndChannelPtr chan, OSType selector,
                    void *infoPtr);

extern pascal OSErr SndSetInfo(
                    SndChannelPtr chan, OSType selector,
                    const void *infoPtr);
```

The two new routines SndGetInfo and SndSetInfo are used to get and set information about the sound environment. Both routines use a selector based interface similar to the SPBGetDeviceInfo and SPBSetDeviceInfo routines found in the Sound Input Manager, and in fact they use the same sound information selectors.

SndGetInfo and SndSetInfo operate on an open Sound Manager channel, and can be used to retrieve and change information about the channel, including hardware settings. These routines should be used instead of attempting to communicate directly with sound components.

These new calls are only available with Sound Manager version 3.1 or later. Check for this by calling SndSoundManagerVersion for the installed version. Note

that you can always open a sound channel for the hardware device that you desire by passing `kUseOptionalOutputDevice` as the synth parameter and the component reference as the init parameter.

```
OSErr OpenChannel(OSType myType)
{
    ComponentDescription    searching;
    Component               outputDevice;
    OSErr                   err;

    // search for a sound output device component
    searching.componentType = kSoundOutputDeviceType;
    searching.componentSubType = myType;
    searching.componentManufacturer = kAnyComponentManufacturer;
    searching.componentFlags = 0;
    searching.componentFlagsMask = kAnyComponentFlagsMask;
    outputDevice = nil;
    outputDevice = FindNextComponent(outputDevice, &searching);

    if (outputDevice == nil)
        err = cantFindHandler; /*component not found*/
    else
    {
        gChan = nil;
        err = SndNewChannel(&gChan, kUseOptionalOutputDevice,
                            (long)outputDevice, nil);
    }
    return (err);
}
```

For example, to determine the current hardware sampling rate of the given sound channel you may use this code:

```
UnsignedFixed sampleRate;
err = SndGetInfo(gChan, siSampleRate, &sampleRate);
GetSoundOutputInfo() and SetSoundOutputInfo()
pascal OSErr GetSoundOutputInfo(Component outputDevice, OSType selector,
                                void *infoPtr);
pascal OSErr SetSoundOutputInfo(Component outputDevice, OSType selector,
                                const void *infoPtr);
```

These two routines get and set information about the sound environment:
`GetSoundOutputInfo` and `SetSoundOutputInfo`. Both routines use a selector based
interface similar to the `SPBGetDeviceInfo` and `SPBSetDeviceInfo` routines found
in the Sound Input Manager, and in fact they use the same sound info selectors.

`GetSoundOutputInfo` and `SetSoundOutputInfo` operate directly on a sound output
device, and can be used to retrieve and change information about the hardware
settings. These routines should be used instead of attempting to communicate
directly with sound output components. Setting the output device parameter to
nil causes the default output device to be used. These calls are similar to
`GetSndInfo` and `SetSndInfo` but do not require an opened sound channel. For
example, to determine the sampling rate of the sound hardware on the default
output device, you could use this code:

```
OSErr GetCurrentSampleRate(UnsignedFixed *sampleRate)
    {

OSErr err;
    err = GetSoundOutputInfo(nil, siSampleRate, sampleRate);
return (err);
}
```

ParseAIFFHeader

```
pascal OSErr ParseAIFFHeader(
                    short fRefNum, SoundComponentData *sndInfo,
                    unsigned long *numFrames,
                    unsigned long *dataOffset);
```

The `ParseAIFFHeader` routine returns information describing the audio data in
the given AIFF file. The `fRefNum` parameter specifies the open AIFF file to use.
The `sndInfo` parameter is a `SoundComponentData` structure that returns the
following information about the format of the sound in the AIFF file:

flags	Always returns 0
format	The sound format (i.e., 'raw', 'twos', 'MAC3', etc.)
numChannels	The number of channels (i.e., 1 = mono, 2 = stereo)
sampleSize	The sample size (i.e., 8 = 8-bit, 16 = 16-bit)
sampleRate	The sampling rate (in samples/second)

sampleCount	The number of audio samples in the file
buffer	Always returns 0
reserved	Always returns 0

The numFrames parameter returns the number of frames of audio data in the file, and the dataOffset parameter returns the byte offset of the first audio sample in the file.

ParseSndHeader

```
pascal OSErr ParseSndHeader(
                SndListHandle sndHandle,
                SoundComponentData *sndInfo,
                unsigned long *numFrames,
                unsigned long *dataOffset);
```

The ParseSndHeader routine returns information describing the audio data in the given 'snd ' resource handle. The sndHandle parameter specifies the sound handle to use. The sndInfo parameter is a SoundComponentData structure that returns the following information about the format of the sound in the handle:

flags	Always returns 0
format	The sound format (i.e., 'raw', 'twos', 'MAC3', etc.)
numChannels	The number of channels (i.e., 1 = mono, 2 = stereo)
sampleSize	The sample size (i.e., 8 = 8-bit, 16 = 16-bit)
sampleRate	The sampling rate (in samples/second)
sampleCount	The number of audio samples in the file
buffer	Always returns 0
reserved	Always returns 0

The numFrames parameter returns the number of frames of audio data in the handle, and the dataOffset parameter returns the byte offset of the first audio sample in the handle.

GetCompressionInfo

```
pascal OSErr GetCompressionInfo(
                    short compressionID, OSType format,
                    short numChannels, short sampleSize,
                    CompressionInfoPtr cp);
```

For a given AIFF file or snd resource, the information contained within it might be used to determine basic characteristics of the sound such as its duration.

```
duration = numSampleFrames / sampleRate
```

Note that this is a valid calculation for an uncompressed sound. But this calculation returns an incorrect result for a compressed sound. The problem here is that each sample frame in a compressed sound is composed of one or more packets rather than sample points (see "Sampled-Sound Data" (page 64)), and each packet in that compressed sound can itself represent several sample points. We therefore need a way to determine the number of samples in a packet in order to get an accurate calculation.

The compressionID parameter defines the compression algorithm used on the sample. The AIFF-C Extended Common Chunk does not contain a compressionID field. In this case (and when using snd resources where the OSType describing the compression format is known) you should always pass the constant fixedCompression in this parameter and the OSType in the format parameter. The format field will then contain the OSType representing the compression format. If you set the compressionID field in a compressed sound header to any value other than fixedCompression, then the format field is set to zero. The format parameter is the OSType describing the format of the compressed sound data. If you pass the constant fixedCompression in the compressionID parameter you will need to pass a valid compression type here. Some of the valid format types are:

```
NONE - sound is not compressed

MAC3 - compression format is MACE 3:1

MAC6 - compression format is MACE 6:1

ima4 - compression format is IMA 4:1
```

There are some snd resources that do not store an OSType in the format field of the compressed sound header describing the compression format. You can still use GetCompressionInfo in this case by passing in the compressionID and passing 0 in the format parameter. The correct OSType will be returned in the format field of the CompressionInfo structure. Using the appropriate fields from an AIFF-C Extended Common Chunk or our snd resource compressed sound header, we can make the call to GetCompressionInfo:

Example Extended Common Chunk

```
myExtendedCommonChunk.ckID = 'COMM';
myExtendedCommonChunk.ckSize = 34;
myExtendedCommonChunk.numChannels = 1;
myExtendedCommonChunk.numSampleFrames = 7633;
myExtendedCommonChunk.sampleSize = 8;
myExtendedCommonChunk.sampleRate = 22254.54545;
myExtendedCommonChunk.compressionType = MAC3;
myExtendedCommonChunk.compressionName = "MACE 3-to-1";
```

Example Compressed Sound Header

```
myCompressedSoundHeader.samplePtr = nil;
myCompressedSoundHeader.numChannels = 1;
myCompressedSoundHeader.sampleRate = rate22khz;
myCompressedSoundHeader.encode = cmpSH;
myCompressedSoundHeader.numFrames = 7633
myCompressedSoundHeader.format = 0;
myCompressedSoundHeader.compressionID = threeToOne;
myCompressedSoundHeader.packetSize = threeToOnePacketSize;
myCompressedSoundHeader.snthID = 0;
myCompressedSoundHeader.sampleSize = 8;

// fill in the CompressionInfo from our Extended Common Chunk

OSErr          err;
CompressionInfo cmpInfo;
```

```
err = GetCompressionInfo(fixedCompression,
                        (OSType)(myExtendedCommonChunk.compressionType),
                        myExtendedCommonChunk.numChannels,
                        myExtendedCommonChunk.sampleSize,
                        &cmpInfo);

// fill in the CompressionInfo from our Compressed Sound Header

OSErr          err;
CompressionInfo cmpInfo;

err = GetCompressionInfo(myCmpSoundHeader->compressionID,
                            myCmpSoundHeader->format,
                            myCmpSoundHeader->numChannels,
                            myCmpSoundHeader->sampleSize,
                            &cmpInfo);
```

Note that this call will work for all sound formats, compressed or
uncompressed.

Using `GetCompressionInfo`, the Sound Manager will do the right thing. You get
back the information you need in the `CompressionInfo` struct, with no special
casing needed. Upon returning from the call to `GetCompressionInfo`, you have a
filled `CompressionInfo` struct.

```
recordSize = 20
format = MAC3
compressionID = threeToOne
samplesPerPacket = 6
bytesPerPacket = 2
bytesPerFrame = 2
bytesPerSample = 1
```

Now you can use this information to determine the duration of our sound.

```
duration = (numSampleFrames * samplesPerPacket) / sampleRate
```

By substituting the data given from the example above, you get the following
results.

```
2.06 seconds = (7633 * 6) / 22254.54545
```

By including the `CompressionInfo` struct you should never need to special case code for compressed vs. uncompressed sounds—and all sound data calculations should be correct.

The following is a list of useful calculations that can be made using the data returned in the struct, along with data from our Extended Common Chunk or Compressed Sound Header:

```
seconds = (numFrames * samplesPerPacket) / sampleRate;
samples = numFrames * samplesPerPacket;
bytes = numFrames * bytesPerFrame;
compressionRatio = (samplesPerPacket * bytesPerSample) / bytesPerPacket;
```

GetCompressionName

```
pascal OSErr GetCompressionName(
                    OSType compressionType,
                    Str255 compressionName);
```

The `GetCompressionName` routine returns a string describing the given compression format in a string that can be displayed to the user. The `compressionType` parameter specifies the compression format, and the name is returned in `compressionName`.

This string can be used in pop-up menus and other user interface elements to allow the user to select a compression format.

SoundConverterGetBufferSizes

```
pascal OSErr SoundConverterGetBufferSizes(
                    SoundConverter sc,
                    unsigned long targetBytes,
                    unsigned long *inputFrames,
                    unsigned long *inputBytes,
                    unsigned long *outputBytes);
```

`SoundConverterGetBufferSizes` is used to determine the input and output buffer sizes for a given target size. This is so you can make sure your buffers will fit the conversion parameters established with `SoundConverterOpen`.

The `targetBytes` parameter is the approximate number of bytes you would like both your input and output buffers to be. The `inputFrames` and `inputBytes` parameters return the actual size you should make your input buffer, in frames and bytes, respectively. The `outputBytes` parameter returns the size in bytes for your output buffer.

Note
The returned input and output buffer sizes can be larger than your target size settings. This is because they are rounded up depending on the format, but they will be very close to the target settings. Also note that the input and output sizes may be very different, depending on the input and output formats given in `SoundConverterOpen`. The sizes are calculated assuming you will convert all data in the input buffer to the output buffer. ◆

SoundConverterBeginConversion

```
pascal OSErr SoundConverterBeginConversion(SoundConverter sc);
```

`SoundConverterBeginConversion` starts a conversion. All state information is reset to default values in preparation for a new input buffer.

This routine can be called at interrupt time.

SoundConverterConvertBuffer

```
pascal OSErr SoundConverterConvertBuffer(
                SoundConverter sc,
                const void *inputPtr, unsigned long inputFrames,
                void *outputPtr, unsigned long *outputFrames,
                unsigned long *outputBytes);
```

SoundConverterConvertBuffer converts a buffer of data from the input format to the output format. The inputPtr parameter points to the input data, and inputFrames gives the number of frames in that buffer. The outputPtr parameter specifies where the output data should be placed. The outputFrames and outputBytes parameters return the number of frames and bytes placed in the output buffer respectively.

This routine will consume all the data in the input buffer. Depending on the complexity of the conversion, however, not all the converted data may be put in the output buffer right away. The SoundConverterEndConversion routine is used to flush out all this remaining data before a conversion session is closed.

If you are using this routine in conjunction with SoundConverterGetBufferSizes, it is very important that you do not pass in a value in inputFrames larger than the frames value returned by SoundConverterGetBufferSizes, or you will overflow your output buffer. The SoundConverterConvertBuffer calls converts ALL the input data! This routine can be called at interrupt time.

SoundConverterEndConversion

```
pascal OSErr SoundConverterEndConversion(
                SoundConverter sc,
                void  *outputPtr,
                unsigned long *outputFrames,
                unsigned long  *outputBytes);
```

SoundConverterEndConversion ends a conversion. Any data remaining in the converters is flushed out and returned here.

This routine can be called at interrupt time.

Glossary

AGC   See **automatic gain control.**

AIFF   See **Audio Interchange File Format.**

AIFF-C   See **Audio Interchange File Format Extension for Compression.**

alert sound   See **system alert sound.**

Alert Sounds control panel   A subpanel of the Sound control panel that allows the user to select a system alert sound. See also **Sound In control panel, Sound Out control panel, Volumes control panel.**

allophone   A distinct variety of a phoneme in a particular language that is never used contrastingly with any other allophone of the phoneme.

amplitude   A modification to the wave amplitude of a sound to make it sound louder or softer. See also **speech volume.** Compare **wave amplitude.**

Apple Mixer   See **Apple Mixer component.**

Apple Mixer component   A sound component that is responsible for mixing together the audio data streams from all open sound channels.

Apple Sound Chip (ASC)   A custom chip that, in conjunction with other circuitry, generates a stereo sound signal that drives the internal speaker or an external sound jack. Compare **Enhanced Apple Sound Chip.**

ASC   See **Apple Sound Chip.**

asynchronous sound play   The playing of sound during other, non-sound related operations. Compare **synchronous sound play.**

audio compression   A technique of reducing the amount of memory space required for a buffer of sampled-sound data, usually at the expense of audio fidelity. See also **audio expansion.**

audio component   A component that works with the Sound Manager to adjust volumes or other settings of a sound output device. Compare **sound component.**

audio data   See **sampled-sound data, sound, square-wave data, wave-table data.**

audio decompression   See **audio expansion.**

audio expansion   The decompression of compressed sound data. See also **audio compression.**

audio information record   A structure you can use to specify information about an audio component. Defined by the `AudioInfo` data type.

Audio Interchange File Format (AIFF)   A sound storage file format designed to allow easy exchange of audio data among applications.

Audio Interchange File Format Extension for Compression (AIFF-C)   An extension of the Audio Interchange File Format that allows for the storage of compressed sound data.

audio port   Any independently-controllable sound-producing hardware connected or attached to a sound output device. A sound output device can have several audio ports.

audio selection record   A structure you can use to specify that only part of a sound be played. Defined by the `AudioSelection` data type.

automatic gain control (AGC)   A feature of sound recording that moderates the recording to give a consistent signal level.

base frequency   The pitch at which a sampled sound is recorded. The wave of a sampled sound may include frequencies other than the base frequency (and need not even include the base frequency).

baseline pitch   See **speech pitch.**

buffered expansion   Audio expansion of a sound that does not occur while the sound is playing. Compare **real-time expansion.**

callback procedure   An application-defined procedure that is invoked at a specified time or based on specified criteria.

channel   A portion of sound data that can be described by a single sound wave. Do not confuse with sound channel or speech channel. See also **monophonic sound, stereo sound.**

chunk   Any distinct portion of a sound file.

chunk header   The first segment of a chunk, which defines the characteristics of the chunk. Defined by the `ChunkHeader` data type.

codec   See **compression/decompression component.**

command   See **embedded speech command, sound command.**

command delimiter   A sequence of one or two characters that indicates the start or end of an embedded speech command.

component   A piece of code that provides a defined set of services to one or more clients. Applications, system extensions, and other components can use the services of a component. See also **audio component, sound component.**

component description record   A structure that contains information about a component. Defined by the `ComponentDescription` data type.

Component Manager   A collection of routines that allows your application or other clients to access components. The Component Manager manages components and also provides services to components.

compressed sound data   Sampled-sound data that has been subjected to audio compression.

compressed sound header   A sound header that can describe noncompressed and compressed sampled-sound data, whether monophonic or stereo. Defined by the `CmpSoundHeader` data type. See also **extended sound header, sampled sound header.**

compression   See **audio compression.**

compression/decompression component (codec)   A component that handles data compression and decompression.

compression information record   A structure you use to specify information about a sound component that can decompress compressed audio data. Defined by the `CompressionInfo` data type.

computer-generated speech   See **synthesized speech.**

continuous play from disk   See **play from disk.**

continuous recording   A feature of a sound input device driver that allows recording from the device while other processing continues.

current sound input device   The sound input device that the user has chosen through the Sound In subpanel of the Sound control panel.

current sound output device   The sound output device that the user has chosen through the Sound Out subpanel of the Sound control panel.

DAC   See **digital-to-analog convertor.**

decompressed sound data   Sampled-sound data that has been subjected to audio compression and expansion.

decompression   See **audio expansion.**

delimiter   See **command delimiter.**

delimiter information record   A structure that defines the characters used to indicate the beginning and end of a command embedded in text. Defined by the `DelimiterInfo` data type.

dictionary   See **pronunciation dictionary.**

digital signal processor (DSP)   A processor that manipulates digital data.

digital-to-analog convertor (DAC)   A device that converts data from digital to analog form.

double buffering   A technique used by the Sound Manager to manage a play from disk. When using this technique, the Sound Manager plays one buffer of sampled-sound data while filling a second with more data. When the first buffer of sound finishes playing, the Sound Manager plays the data in the second buffer while filling the first with more data. See also **play from disk, sampled-sound data**.

drop-sample conversion   A form of sample rate conversion that uses an existing sample as an interpolated sample point. Compare **linear interpolation.**

DSP   See **digital signal processor.**

duration   The length of time that a sound takes to play.

EASC   See **Enhanced Apple Sound Chip.**

embedded speech command   In a buffer of input text, a sequence of characters enclosed by command delimiters that provides instructions to a speech synthesizer.

ending prosody   The rhythm, modulation, and stress patterns associated with the end of a sentence of speech.

Enhanced Apple Sound Chip (EASC)   A modified Apple Sound Chip that generates stereo sound using pulse-code modulation. Compare **Apple Sound Chip.**

enhanced Sound Manager   Any version of the Sound Manager greater than 2.0.

error callback procedure   An application-defined procedure that is executed whenever the Speech Manager encounters an error in an embedded speech command in a buffer of input text.

expansion   See **audio expansion.**

extended sound header   A sound header that can describe monophonic and stereo sampled-sound data, but not compressed sound data. Defined by the `ExtSoundHeader` data type. See also **compressed sound header, sampled sound header**.

FIFO   See **first-in, first-out.**

Finder sound file   A file of file type `'sfil'` containing a sound resource. If a user opens a Finder sound file, the Finder plays the sound resource contained within it. See also **sound file, sound resource.**

first-in, first-out (FIFO)   Characteristic of a queue in which the first item put into the queue becomes the first item to be taken out of it. Compare **last-in, first out.**

frequency   The number of times per second that an action occurs. An action's frequency is measured in cycles per second, or hertz. See also **period.**

gain   The ratio of the output volume to the input volume. See also **automatic gain control.**

hertz (Hz)   A unit of frequency, equal to one cycle per second.

instrument   A sampled sound played at varying rates to produce a number of different pitches or notes. See also **voice.**

interleaving   The technique of combining two or more channels of sound data by alternating small pieces of the data in each channel into a single data stream. See also **sample frame.**

interpolation   The process of generating sample points between two given sample points. See also **linear interpolation.**

kilohertz (kHz)   A unit of frequency, equal to one thousand cycles per second.

last-in, first out (LIFO)   Characteristic of a queue in which the last item put into the queue becomes the first item to be taken out of it. Compare **first-in, first out.**

LIFO   See **last-in, first-out.**

linear interpolation   A form of interpolation that uses the calculated mean of two sample points as the interpolated sample point. Compare **drop-sample conversion.**

MACE   See **Macintosh Audio Compression and Expansion.**

Macintosh Audio Compression and Expansion (MACE)   A set of Sound Manager routines that allow your application to compress and expand audio data.

megahertz (MHz) A unit of frequency, equal to one million cycles per second.

microsecond A unit of time equal to one millionth of a second. Abbreviated μs.

MIDI See **Musical Instrument Digital Interface.**

MIDI Manager The part of the Macintosh system software that controls the flow of MIDI data and commands through a MIDI interface.

MIDI note value An integer that is defined to correspond to a frequency specified in hertz that is associated with a musical note.

millisecond A unit of time equal to one thousandth of a second. Abbreviated ms.

modulation of speech See **pitch modulation.**

monophonic sound. Sound consisting of a single channel. Compare **stereo sound.**

multichannel sound See **stereo sound.**

Musical Instrument Digital Interface (MIDI) A standard protocol for sending audio data and commands to digital devices.

noncompressed sound data Sampled-sound data that has not been subjected to audio compression or that has been decompressed.

note See **frequency, MIDI note value.**

offset-binary encoding A method of digitally encoding sound that represents the range of amplitude values as an unsigned number, with the midpoint of the range representing silence. For example, an 8-bit sound stored in offset-binary format would contain sample values ranging from 0 to 255, with a value of 128 specifying silence (no amplitude). Samples in Macintosh sound resources are stored in offset-binary form. See also **two's complement encoding.**

packet A unit of compressed sampled-sound data. One or more packets make up a sample frame of compressed sampled-sound data. See also **sample point.**

period The time elapsed during one complete cycle. See also **frequency.**

phoneme A speech sound in a language that a speaker of the language psychologically considers to be a single unit. A single phoneme may have several allophones.

phoneme callback procedure An application-defined procedure that is executed whenever the Speech Manager is about to pronounce a phoneme.

phoneme descriptor record A structure that contains information about all phonemes defined for the current synthesizer. Defined by the `PhonemeDescriptor` data type.

phoneme information record A structure that contains information about a phoneme. Defined by the `PhonemeInfo` data type.

phonemic representation of speech The representation of speech using a series of phonemes.

phonetic representation of speech The representation of speech using a series of allophones.

pitch A listener's subjective interpretation of a sound's frequency. See also **speech pitch.**

pitch modulation A fixed-point value defined on a scale from 0.000 to 100.000 that indicates the maximum amount by which the frequency of generated speech may deviate from that corresponding to the speech pitch in either direction. A value of 0.000 corresponds to a monotone.

play from disk The ability of the Sound Manager to play sampled sounds stored on disk (either in a sound file or a sound resource) continuously without audible gaps.

playthrough A feature of sound recording that allows the user to hear, through the speaker of a Macintosh computer, the sound being recorded.

polyphonic sound See **stereo sound.**

pronunciation dictionary A list of words and their pronunciations, installed in a speech channel to override default speech synthesizer pronunciations of words.

pronunciation dictionary resource A pronunciation dictionary stored in a resource of type `'dict'`.

prosody   The rhythm, modulation, and stress patterns of speech.

rate   See **sample rate, speech rate**.

real-time expansion   Audio expansion of a sound that occurs while the sound is playing. Compare **buffered expansion.**

recording   The process of creating an analog or digital representation of a sound. See also **sampling.**

sample   See **sample point.**

sampled sound   Any sound defined using sampled-sound data.

sampled-sound data   Any set of values that represent the sample points of a sampled sound. The values can be in either offset-binary format or two's complement format.

sampled sound header   A sound header that can describe monophonic, noncompressed sampled-sound data. Defined by the `SoundHeader` data type. See also **compressed sound header, extended sound header.**

sample frame   An interleaved set of sample points (for noncompressed sampled-sound data) or packets (for compressed sampled-sound data).

sample point   A value representing the amplitude of sampled-sound data at a particular instant. One or more sample points make up a sample frame of noncompressed sampled-sound data. See also **packet.**

sample rate   The rate at which samples are recorded. Sample rates are usually measured in kilohertz or megahertz.

sampling   The process of representing a sound by measuring its amplitude at discrete points in time. See also **recording.**

sifter   See **sound component.**

sound   Anything perceived by the organs of hearing. See also **frequency, pitch, stereo sound, timbre.**

sound channel   A path that sound data traverses from an application to the sound output device. A sound channel is associated with a queue of sound commands and with other information about the audio characteristics of the sound data. See also **sound channel record.**

sound channel record   A structure that represents a sound channel. Defined by the `SndChannel` data type.

sound channel status record   A structure whose address you pass to the `SndChannelStatus` function. Defined by the `SCStatus` data type.

sound command   An instruction to produce sound, modify sound, or otherwise assist in the overall process of sound production. See also **sound command record.**

sound command record   A structure that describes a sound command. Defined by the `SndCommand` data type.

sound component   A component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. See also **audio component, compression/ decompression component, sound output device component, utility component.**

sound component chain   A chain of sound components that links a sound source to a sound output device.

sound component data record   A structure that specifies information about the data stream generated by a sound component. Defined by the `SoundComponentData` data type.

sound component information selector   A value of type `OSType` that indicates the kind of information a sound component should return or modify.

Sound control panel   A control panel that allows the user to specify basic sound-related settings and preferences. See also **Alert Sounds control panel, Sound In control panel, Sound Out control panel, Volumes control panel.**

sound data   See **sampled-sound data, sound, square-wave data, wave-table data.**

sound double buffer header record   A structure that you use to manage your own double-buffering scheme. Defined by the `SndDoubleBufferHeader` and `SndDoubleBufferHeader2` data types.

sound double buffer record   A structure that you use to manage your own double-buffering scheme. Defined by the `SndDoubleBuffer` data type.

Sound Driver   A device driver on the original Macintosh computers that provided sound generation. The Sound Driver is now obsolete; it has been replaced by the Sound Manager.

sound file   A file of file type `'AIFF'` or `'AIFC'` that can be used to store sampled-sound data and information about that data. See also **Audio Interchange File Format, Audio Interchange File Format Extension for Compression, chunk, Finder sound file, sound resource.**

sound header   A data structure (usually stored in a sound resource) that contains information about a buffer of sampled-sound data. See also **compressed sound header, extended sound header, sampled sound header.**

Sound In control panel   A subpanel of the Sound control panel that allows the user to select a sound input device. See also **Alert Sounds control panel, Sound Out control panel, Volumes control panel.**

sound information list   A structure that specifies the information associated with a sound component information selector. Defined by the `SoundInfoList` data type.

sound input device   Any hardware device (such as a microphone or audio digitizer) that records sound.

sound input device driver   A standard Macintosh device driver used by the Sound Manager to manage communication between applications and a sound input device.

sound input device information selector   A variable of type `OSType` that is used to specify the type of information that an application or the Sound Input Manager is requesting from a sound input device driver.

Sound Input Manager   The part of the Macintosh system software that controls the recording of sound from sound input devices.

sound input parameter block   A parameter block that contains information about sound recording. Defined by the `SPB` data type.

Sound Manager   The part of the Macintosh system software that manages the production and manipulation of sounds on Macintosh computers.

Sound Manager status record   A structure filled in by the `SndManagerStatus` function, which gives information on the current CPU loading caused by all open channels of sound. Defined by the `SMStatus` data type.

Sound Out control panel   A subpanel of the Sound control panel that allows the user to select a sound output device. See also **Alert Sounds control panel, Sound In control panel, Volumes control panel.**

sound output device   Any hardware device (such as a speaker or sound synthesizer) that produces sound.

sound output device component   A sound component that communicates with a sound output device. See also **compression/decompression component** and **utility component.**

sound parameter block   A parameter block that describes the source data to be modified or sent to a sound output device. Defined by the `SoundParamBlock` data type.

sound recording dialog box   The dialog box displayed by the Sound Input Manager when you call `SndRecord` or `SndRecordToFile`.

sound resource   A resource of resource type `'snd '` that can be use to store sound commands and sound data. See also **sound file.**

sound resource header   The portion of a sound resource that describes the format of the sound resource.

sound source   The origin of a specific channel of sound.

source   See **sound source.**

source component   The sound component that provides input for a particular component.

source ID   A unique 4-byte identifier created by the Apple Mixer to refer to a single chain of sound components linking a sound source to the current sound output device. Defined by the `SoundSource` data type.

speech   The process or product of speaking. See also **sound, synthesized speech.**

speech amplitude   See **speech volume.**

speech attribute   A setting defined for a voice or a class of voices that affects the quality of speech generated by the Speech Manager. Speech attributes include **speech pitch, speech rate, pitch modulation, speech volume.**

speech channel   The data structure used by the Speech Manager to store settings related to speech generation. All speech must be generated through a speech channel. Defined by the `SpeechChannel` data type.

speech channel control flags   Constants that enable special Speech Manager features associated with speech generation.

speech command   See **embedded speech command.**

speech-done callback procedure   An application-defined procedure that is executed when the Speech Manager completes speaking a buffer of input text.

speech error information record   A structure that contains information about which Speech Manager errors occurred while processing a text buffer on a given speech channel. Defined by the `SpeechErrorInfo` data type.

speech extension data record   A structure passed to `GetSpeechInfo` or `SetSpeechInfo` to get or set synthesizer information. Defined by the `SpeechXtndData` data type.

speech information selector   A variable of type `OSType` that is used to specify the type of information that an application or the Speech Manager is requesting from a speech synthesizer.

Speech Manager   The part of the Macintosh system software that provides a standardized method for Macintosh applications to generate synthesized speech.

speech modulation   See **pitch modulation.**

speech pitch   A fixed-point value on a scale from 0.000 to 100.000 that indicates the average (or baseline) frequency a speech synthesizer should use in generating synthesized speech. A value of 60.000 corresponds to Middle C on a conventional piano keyboard. See also **pitch modulation.**

speech rate   A fixed-point value specifying the approximate number of words per minute that a speech synthesizer should use in generating speech.

speech status information record   A structure that contains information about the status of a speech channel. Defined by the `SpeechStatusInfo` data type.

speech synthesizer   The executable code that is linked to a speech channel and manages all communication between the Speech Manager and the Sound Manager.

speech version information record   A structure that contains information about the speech synthesizer currently being used. Defined by the `SpeechVersionInfo` data type.

speech volume   A fixed-point value on a scale from 0.000 to 1.000 that indicates the average amplitude a speech synthesizer should use in generating synthesized speech. A value of 0.000 corresponds to the lowest possible volume, and a value of 1.000 corresponds to the highest.

square-wave data   Any set of values that represent a sound by its frequency, amplitude, and duration.

stereo sound   Sound that simultaneously consists of two or more channels. Also called *polyphonic sound* or *multichannel sound.* Compare **monophonic sound.**

synchronization callback procedure   An application-defined procedure that is executed whenever the Speech Manager encounters an embedded synchronization speech command in a buffer of input text.

synchronous sound play   A playing of sound by the Sound Manager that prevents other code from executing until the sound is done playing. Compare **asynchronous sound play.**

synthesized speech   The product of converting nonaural tokens (such as written or digitally-stored words or phonemes) into speech. See also **Speech Manager.**

synthesizer   See **speech synthesizer.**

system alert sound   A sound resource stored in the System file that is played whenever an application or other executable code calls the `SysBeep` procedure.

text   The written representation of language.

text-done callback procedure   An application-defined procedure that is executed when the Speech Manager has finished processing (although not necessarily speaking) a buffer of input text.

text-to-speech   See **synthesized speech.**

tick   A unit of time equal to one sixtieth of a second.

timbre   The tone of a sound, which can range from clear to buzzing.

two's complement encoding   A system for digitally encoding sound that stores the amplitude values as a signed number—silence is represented by a sample with a value of 0. For example, with 8-bit sound samples, two's complement values would range from –128 to 127, with 0 meaning silence. The Audio Interchange File Format (AIFF) used by the Sound Manager stores samples in two's complement form. Compare **offset-binary encoding.**

uncompressed sound data   See **decompressed sound data, noncompressed sound data.**

utility component   A sound component that performs some modification on sound data and does not communicate directly with any sound output device. See also **sound component, sound output device component.**

version record   A structure that contains version information. Defined by the `NumVersion` data type.

voice   (1) The set of parameters that specify a particular quality of synthesized speech. A voice is designed to work with a particular speech synthesizer. (2) A sampled sound played at varying rates to produce a number of different pitches or notes. See also **instrument.**

voice description record   A structure that contains information about a voice. Defined by the `VoiceDescription` data type.

voice file information record   A structure that contains information about the file in which a voice is stored and the resource ID of the voice within that file. Defined by the `VoiceFileInfo` data type.

voice specification record   A structure that provides a unique specification that you must use to obtain information about a voice. Defined by the `VoiceSpec` data type.

volume   See **amplitude, speech volume.**

Volumes control panel   A subpanel of the Sound control panel that allows the user to select volumes. See also **Alert Sounds control panel, Sound In control panel, Sound Out control panel.**

VOX recording   A feature that allows sound recording only when the sound to be recorded exceeds a certain amplitude.

VOX stopping   A feature that stops sound recording when the sound falls below a certain amplitude.

wave amplitude   The height of a sound wave at an instant of time. Compare **amplitude.**

waveform   The shape of a wave (a graph of a wave's amplitude over time).

wavelength   The extent of one complete cycle of a wave.

wave table   A sequence of wave amplitudes measured at fixed intervals.

wave-table data   Any set of values that represent a sound by a wave table.

word callback procedure   An application-defined procedure that is executed whenever the Speech Manager is about to speak a word.

Index

C

callBackCmd command
  described 149
  example of use 103
  using to synchronize sound with other actions 106
callback procedures
  installing 103
  and Sound Manager 101 to 106, 207 to 208
channels. *See* sound channels, speech channels
`ChunkHeader` data type 138, 168
chunk header record 138
chunk headers 168
chunks (in AIFF and AIFF-C files)
  Annotation 137
  Application Specific 137
  Audio Recording 137
  Author 137
  Comments 137
  Common 137, 140 to 142, 170 to 172
  Copyright 137
  data types used to describe 138
  defined 136
  determining size of 119
  Extended Common Chunks 140, 170 to 172
  finding 117 to 121
  Form 137, 138 to 139, 168 to 169
  Format Version 137, 139 to 140, 169
  IDs for 153 to 154
  Instrument 137
  list of types 137
  local 139, 169
  Marker 137
  MIDI Data 137
  modifying 143
  Name 137
  order of 143
  Sound Accelerator 137
  Sound Data 137, 142, 172 to 173
  structure of 137 to 138
`clockComponentCmd` function 335
`CloseMixerSoundComponent` function 299
`CmpSoundHeader` data type 163 to 166
codecs. *See* compression/decompression components
commands. *See* embedded speech commands, sound
    commands
Comments Chunks 137
`CommonChunk` data type 140, 170
Common Chunks 137, 140 to 142, 170
`Comp3to1` procedure 121, 198 to 199
`Comp6to1` procedure 121, 199 to 200
completion routines
  and Sound Input Manager 220, 264 to 265
  and Sound Manager 102, 107, 206 to 207
`ComponentDescription` data type 274

Component Manager
  and sound components 267
  and Speech Manager 34
`ComponentResource` data type 273 to 276
components. *See* audio components, sound components
component selectors 277 to 279
compressed sound header records 163 to 166
compression. *See* audio compression
compression/decompression components (codecs) 22,
    28 to 29, 267, 270, 294
compression IDs 165
`CompressionInfo` data type 297
compression information records 297
compression types 141, 171, 172
computer-generated speech. *See* Speech Manager
`ContainerChunk` data type 139, 169
container chunks. *See* Form Chunks
continuous play from disk. *See* play-from-disk routines
continuous recording
  defined 215
  supporting 227
Control calls 216, 223 to 226
Copyright Chunks 137
CPU loading values 95
cultural values, associated with sounds 38
current sound input device 30
current sound output device 25

D

DAC. *See* digital-to-analog convertor
data
  See also audio data
  sampled-sound 64 to 66
  square-wave 62
  wave-table 63
data format flags 292 to 294
data offset bit in sound commands 130
decompression. *See* audio expansion
delimiter. *See* command delimiter
Device Manager, and sound input device drivers 223
    to 227
dictionaries. *See* pronunciation dictionaries
digital signal processor (DSP) 21, 23, 24, 269
digital-to-analog converter (DAC) 24
document annotations, audio 38
doubleback procedures
  defined 127 to 128, 208 to 209
  limitations of 127
  and sound double buffer header records 167
  syntax of 127
  writing 127 to 128
double buffering 33