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, the Sound Input Manager to record sounds, and the Speech Manager to convert text strings into spoken words.

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, Sound Input Manager, and Speech 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

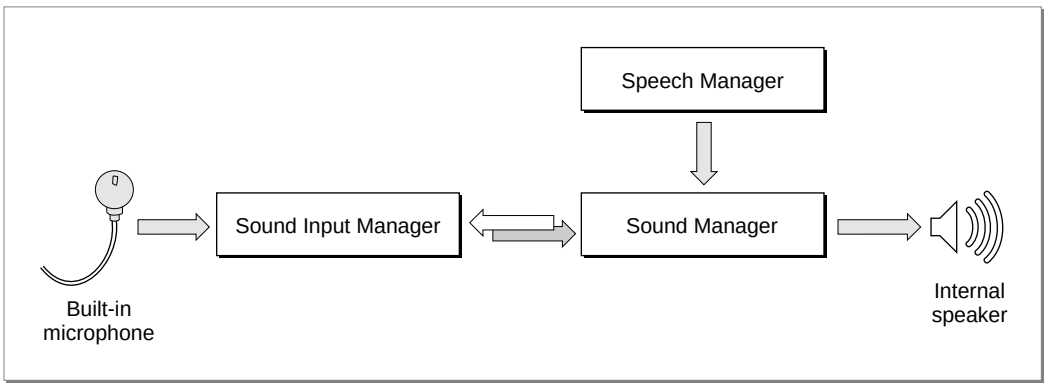
The Macintosh hardware and system software provide a standard and extensible set of capabilities for producing and recording sounds. No matter what kind of application you are developing, you can use these capabilities to enrich your application, often at very little programming expense. For example, 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:

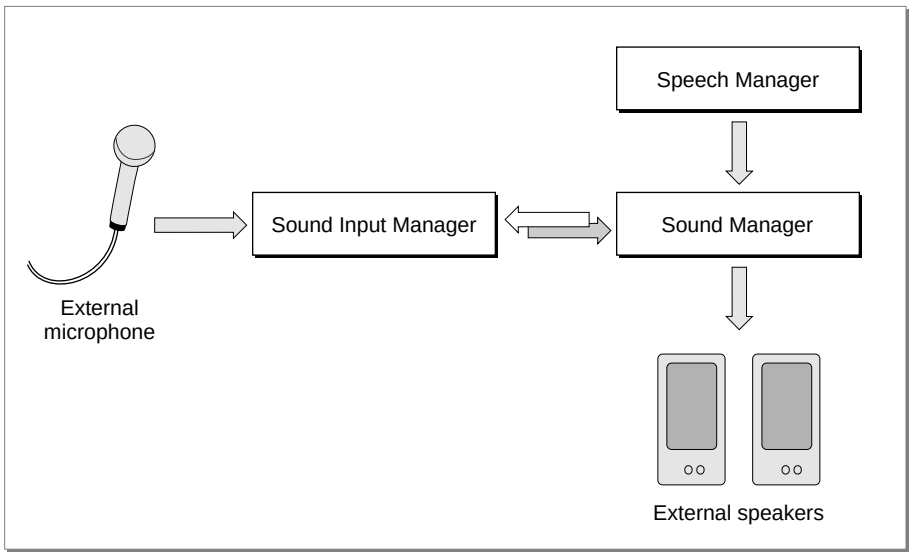
- The **Sound Manager** provides the ability to play sounds through the speaker. It also provides an extensive set of tools for manipulating sounds. You can use the Sound Manager to alter virtually any characteristic of a sound, such as its loudness, pitch, timbre, and duration. You can also use the 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 1-17) 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.

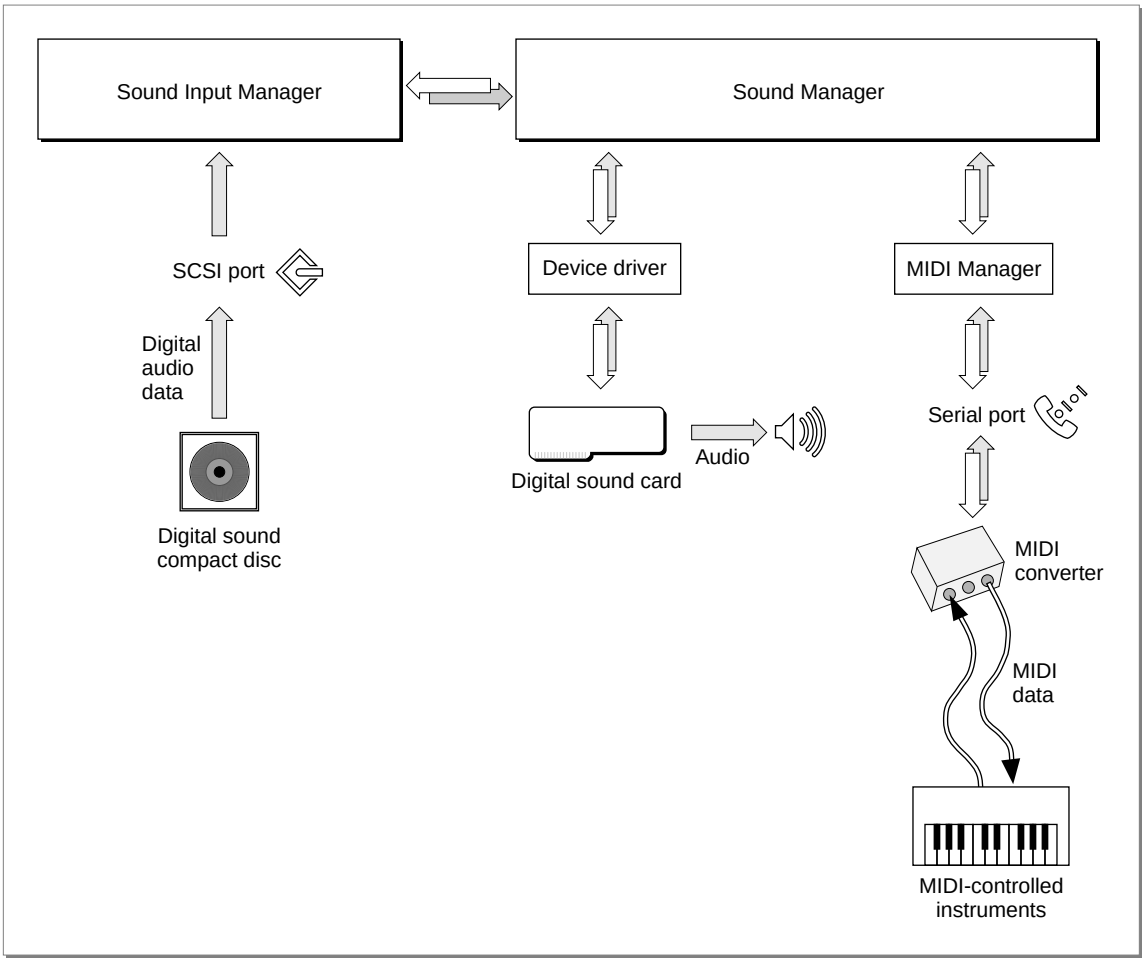
It's possible to enhance the audio capabilities of a Macintosh computer even further. For example, a user can add a NuBus™ expansion card that contains very high quality digital signal processing (DSP) circuitry, together with sound input or output hardware. These cards typically bypass the standard 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

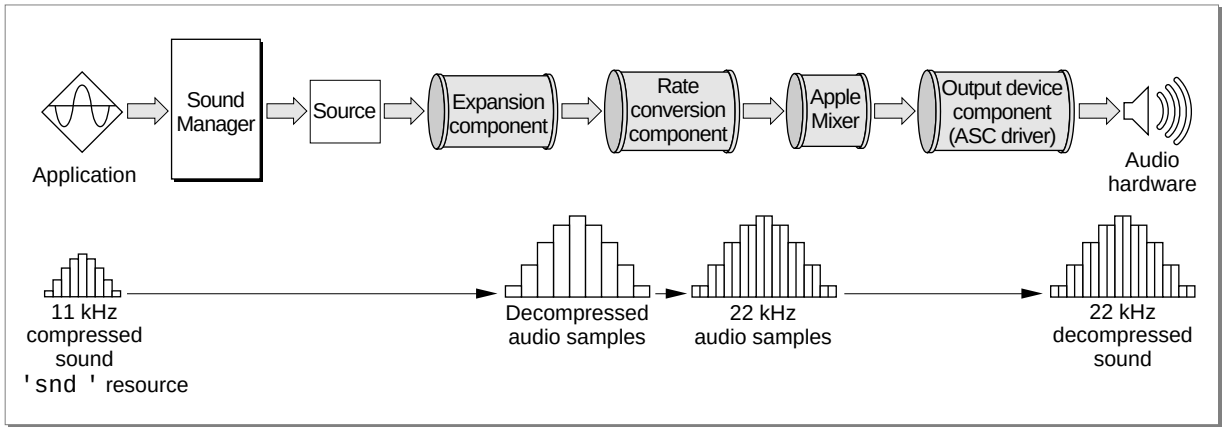


It's possible to enhance the sound environment on a Macintosh computer by adding software alone, for example by adding custom sound **compression/decompression components (codecs).** Apple Computer supplies codecs that can handle 3:1 and 6:1 compression and expansion, which are suitable for most audio 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

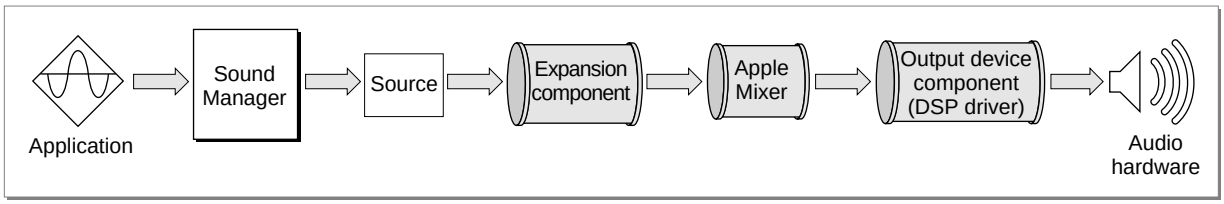
perform one or more signal-processing operations on sound data. For example, the Sound Manager includes sound components for compressing and decompressing sound data (as described in the previous paragraph) and for converting sample rates. Sound components have a standard programming interface and local 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.

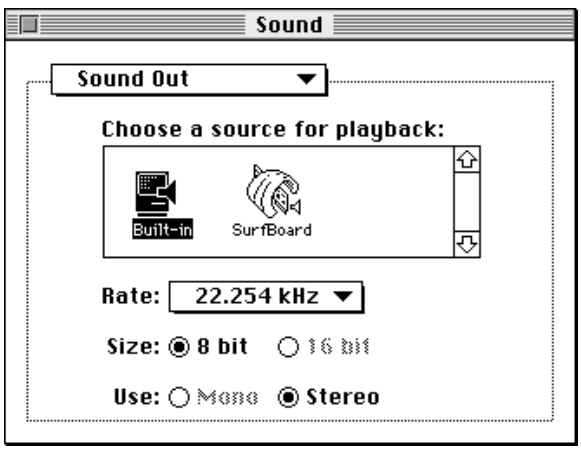
The DAC in Macintosh Plus and Macintosh SE computers is a Sony sound chip. The Macintosh II, Macintosh Portable, Macintosh PowerBook and Macintosh Quadra

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



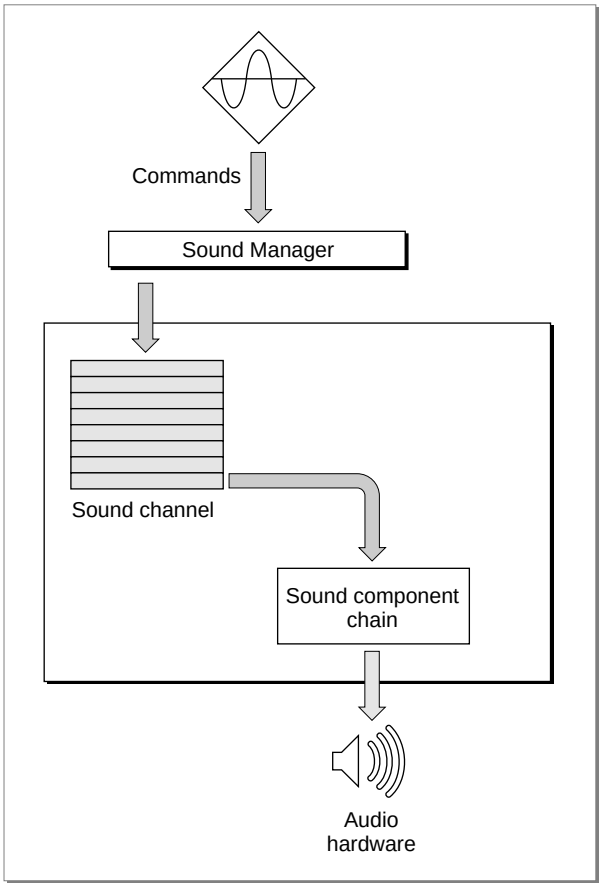
Note
This book shows the Sound control panels introduced with version 3.0 of the Sound Manager. Users can use the pop-up menu at the top of the panel to select one of four or more subpanels (Alert Sounds, Sound In, Sound Out, and Volumes). It's possible to add new subpanels to the Sound control panel. 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



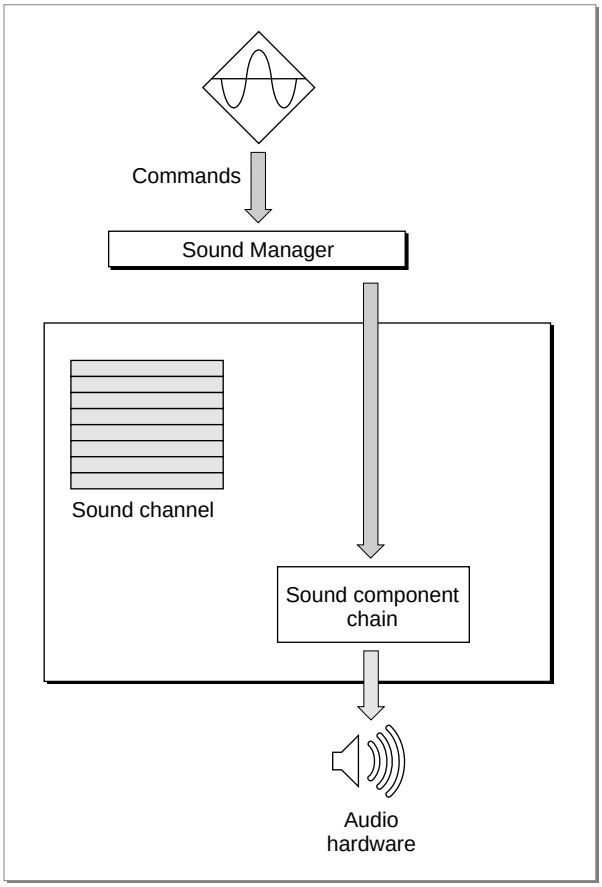
Note
This chapter does not discuss sound commands or channels in detail, because you do not need to know about these details to play sound data stored in sound resources or sound files. This chapter describes only how to play and record sampled sounds. For more information on sound channels and sound 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.

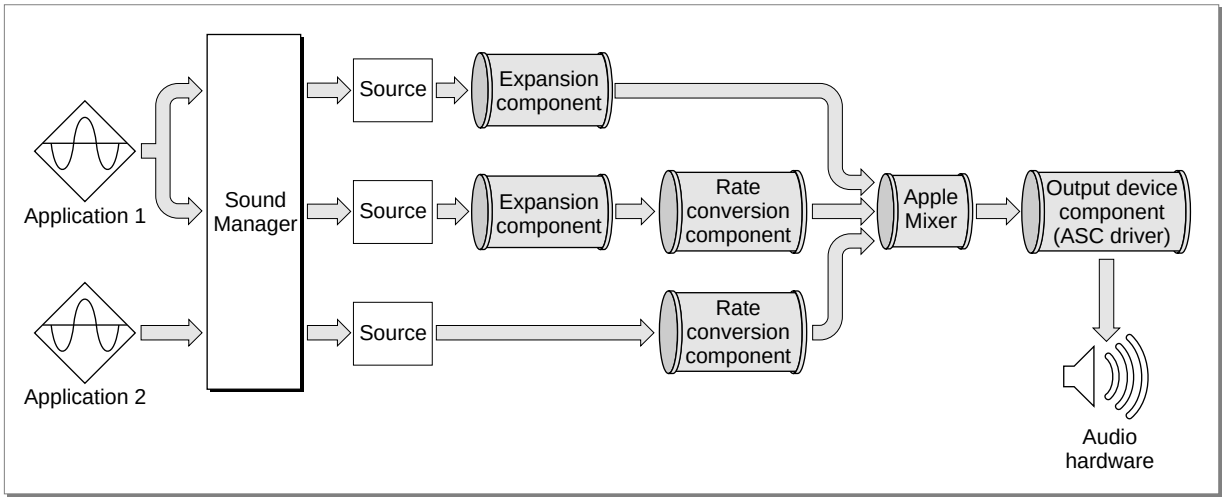
When you bypass the sound channel in this way, any commands that are already queued but not yet sent to the sound output device component remain queued. You can, however, flush the channel at any time by sending the Sound Manager the appropriate request.

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 1-27 and in "Checking For Speech Capabilities" beginning on page 1-31.

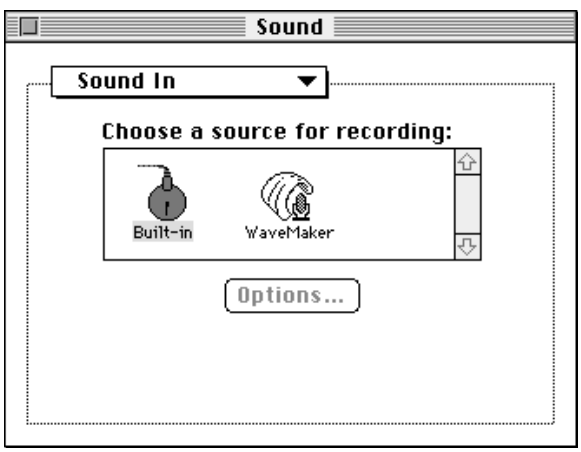
This book describes the capabilities and programming interfaces of version 3.0 of the Sound Manager. Many of the techniques described here can also be used with earlier versions of the Sound Manager, but some cannot. Make sure to test your application thoroughly with all versions of the Sound Manager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

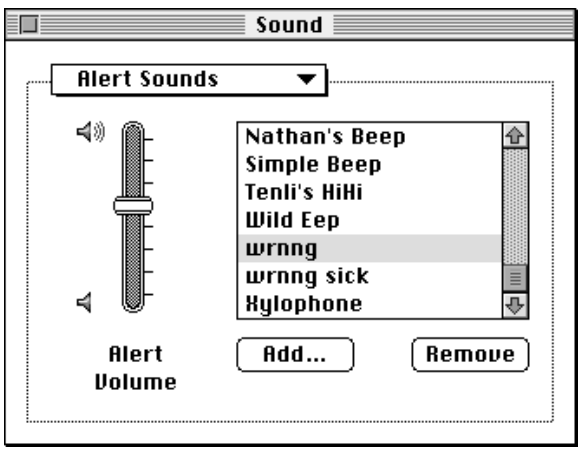
Figure 1-10 The Sound In control panel



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

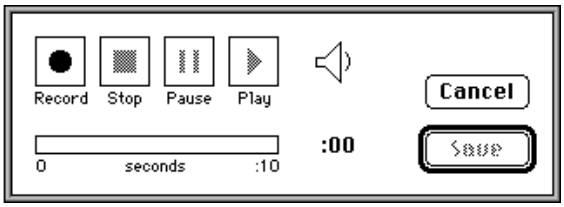


The Alert Sounds control panel also includes two buttons, Add and Remove. These buttons allow the user to add sounds to and remove sounds from the list of available system alert sounds. The Add button is used to record a new alert sound and add it to the list. Clicking the Add button causes the Sound 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.

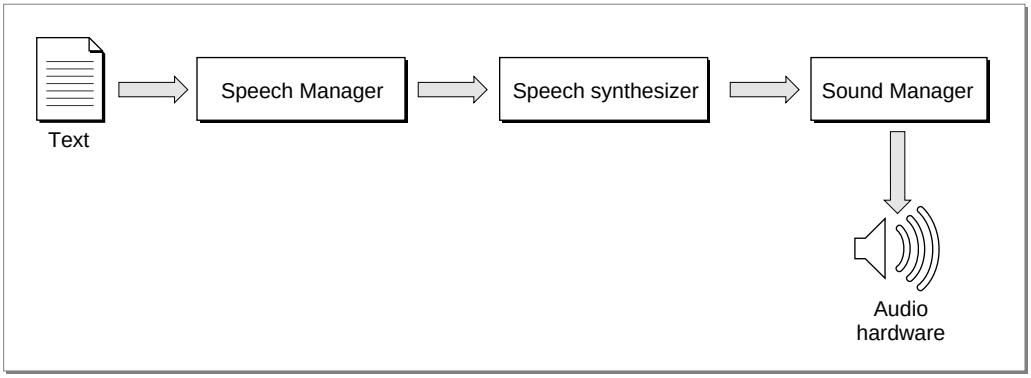
Speech Generation

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.

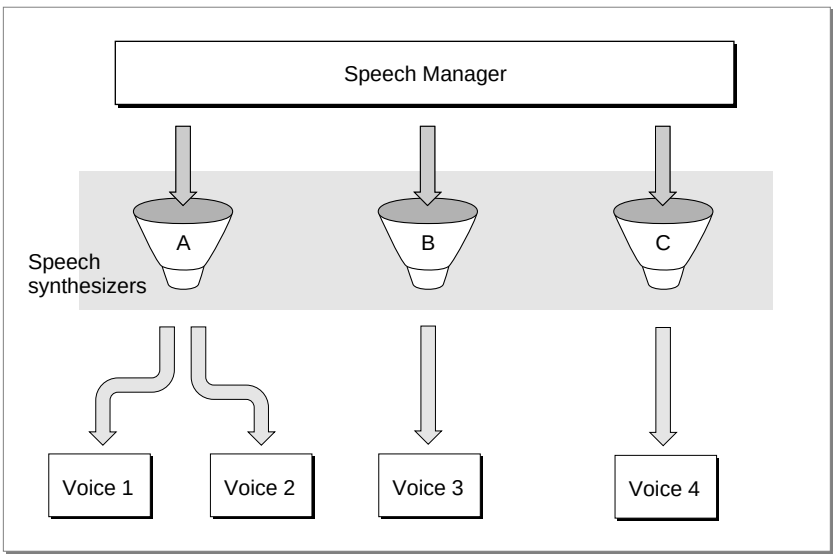
Figure 1-13 The speech generation process



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 1-32, 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

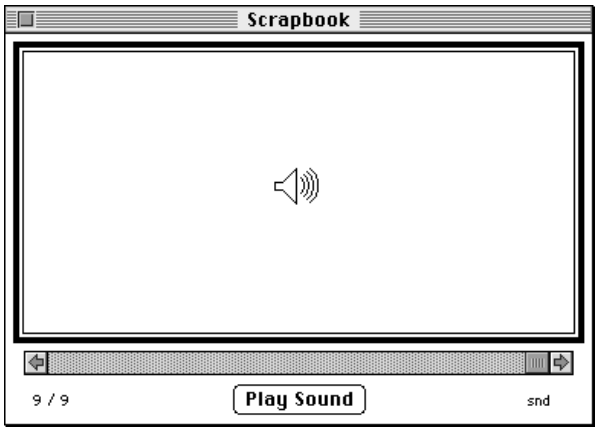
The system software has already defined a set of user interface elements and metaphors that are designed to facilitate the integration of sound into the Macintosh graphical user interface. In general, you should use the existing system software services to present the standard interface elements designed 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 1-17). 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 1-16). 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

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 1-28. 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 1-26 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 1-25. 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 1-26. 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 1-28. 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 1-39.

`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 1-31. 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 1-32. 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

Summary of Sound

Pascal Summary

Constants

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

   {Gestalt selector and response bits for speech attributes}
   gestaltSpeechAttr          = 'ttsc';{speech attributes selector}
   gestaltSpeechMgrPresent    = 0;     {Speech Manager is present}

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

Routines

Playing Sounds

```
PROCEDURE SysBeep           (duration: Integer);
FUNCTION SndPlay            (chan: SndChannelPtr; sndHdl: Handle;
                             async: Boolean): OSErr;
```

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

Recording Sounds

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

Generating and Stopping Speech

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

C Summary

Constants

```
/*Gestalt sound attributes selector and response bits*/
#define gestaltSoundAttr    'snd '  /*sound attributes selector*/

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

```
/*Gestalt selector and response bits for speech attributes*/
#define gestaltSpeechAttr  'ttsc'   /*speech attributes selector*/
enum {
   gestaltSpeechMgrPresent    = 0   /*Speech Manager is present*/
};

/*recording qualities*/
#define siBestQuality         'best'   /*the best quality available*/
#define siBetterQuality       'betr'   /*a quality better than good*/
#define siGoodQuality         'good'   /*a good quality*/
```

Routines

Playing Sounds

```
pascal void SysBeep         (short duration);
pascal OSErr SndPlay        (SndChannelPtr chan, Handle sndHdl,
                             Boolean async);
pascal OSErr SndStartFilePlay
                            (SndChannelPtr chan, short fRefNum,
                             short resNum, long bufferSize, void *theBuffer,
                             AudioSelectionPtr theSelection,
                             FilePlayCompletionProcPtr theCompletion,
                             Boolean async);
```

Recording Sounds

```
pascal OSErr SndRecord      (ModalFilterProcPtr filterProc, Point corner,
                             OSType quality, Handle *sndHandle);
pascal OSErr SndRecordToFile
                            (ModalFilterProcPtr filterProc, Point corner,
                             OSType quality, short fRefNum);
```

Generating and Stopping Speech

```
pascal OSErr SpeakString    (StringPtr s);
pascal short SpeechBusy     (void);
```

Result Codes

`noErr`	0	No error
`userCanceledErr`	–128	User canceled the operation
`noHardwareErr`	–200	Required sound hardware not available
`notEnoughHardwareErr`	–201	Insufficient hardware available
`queueFull`	–203	No room in the queue
`resProblem`	–204	Problem loading the resource
`badChannel`	–205	Channel is corrupt or unusable
`badFormat`	–206	Resource is corrupt or unusable
`notEnoughBufferSpace`	–207	Insufficient memory available
`badFileFormat`	–208	File is corrupt or unusable, or not AIFF or AIFF-C
`channelBusy`	–209	Channel is busy
`buffersTooSmall`	–210	Buffer is too small
`siBadSoundInDevice`	–221	Invalid sound input device
`siInvalidCompression`	–223	Invalid compression type
`siUnknownQuality`	–232	Unknown quality
`synthOpenFailed`	–241	Could not open another speech synthesizer channel