

22

SOUND

Includes Demonstration Program Sound

Introduction to Sound

On the Macintosh, the hardware and software aspects of producing and recording sounds are very tightly integrated.

Audio Hardware

The **audio hardware** includes an internal speaker, a microphone, and one or more integrated circuits that convert digital data to analog signals and analog signals to digital data. The actual integrated circuits that perform these conversions vary between different models of Macintosh computers.

Sound-Related System Software

The sound-related system software managers are as follows:

- ***The Sound Manager.*** The Sound Manager provides the ability to:
 - Play sounds through the speaker.
 - Manipulate sounds, that is, vary such characteristics as loudness, pitch, timbre, and duration.
 - 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.*** The Sound Input Manager provides the ability to record sounds through a microphone or other sound input device.
- ***The Speech Manager.*** The Speech Manager provides the ability to convert written text into spoken words.

Sound Input and Output Capabilities

The basic audio hardware, together with the sound-related system software, provides for the following sound input and output capabilities:

- 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 synchronisation of multiple channels of sampled sounds.
- Compression and decompression of sound data to minimise storage space.

The basic audio hardware and system software also provide the ability to integrate and synchronise 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.

Monitors and Sound Control Panel. For playback, the user can select a sound output device, and set certain characteristics of the selected device, using the Monitors and Sound control panel. The Monitors and Sound control panel also allows the user to select the input device for recording sounds.

Basic and Enhanced Sound Capabilities

It's very easy for users to enhance the quality of the sounds they play back or record by substituting different speakers and microphones for the ones built into a Macintosh computer. Audio capabilities may be further enhanced by adding an expansion card containing very high quality **digital signal processing (DSP)** circuitry, together with sound input or output hardware. Another enhancement option is to add a **MIDI interface** to one of the serial ports. Fig 1 illustrates the basic sound capabilities of the Macintosh and how those capabilities may be further enhanced and extended.

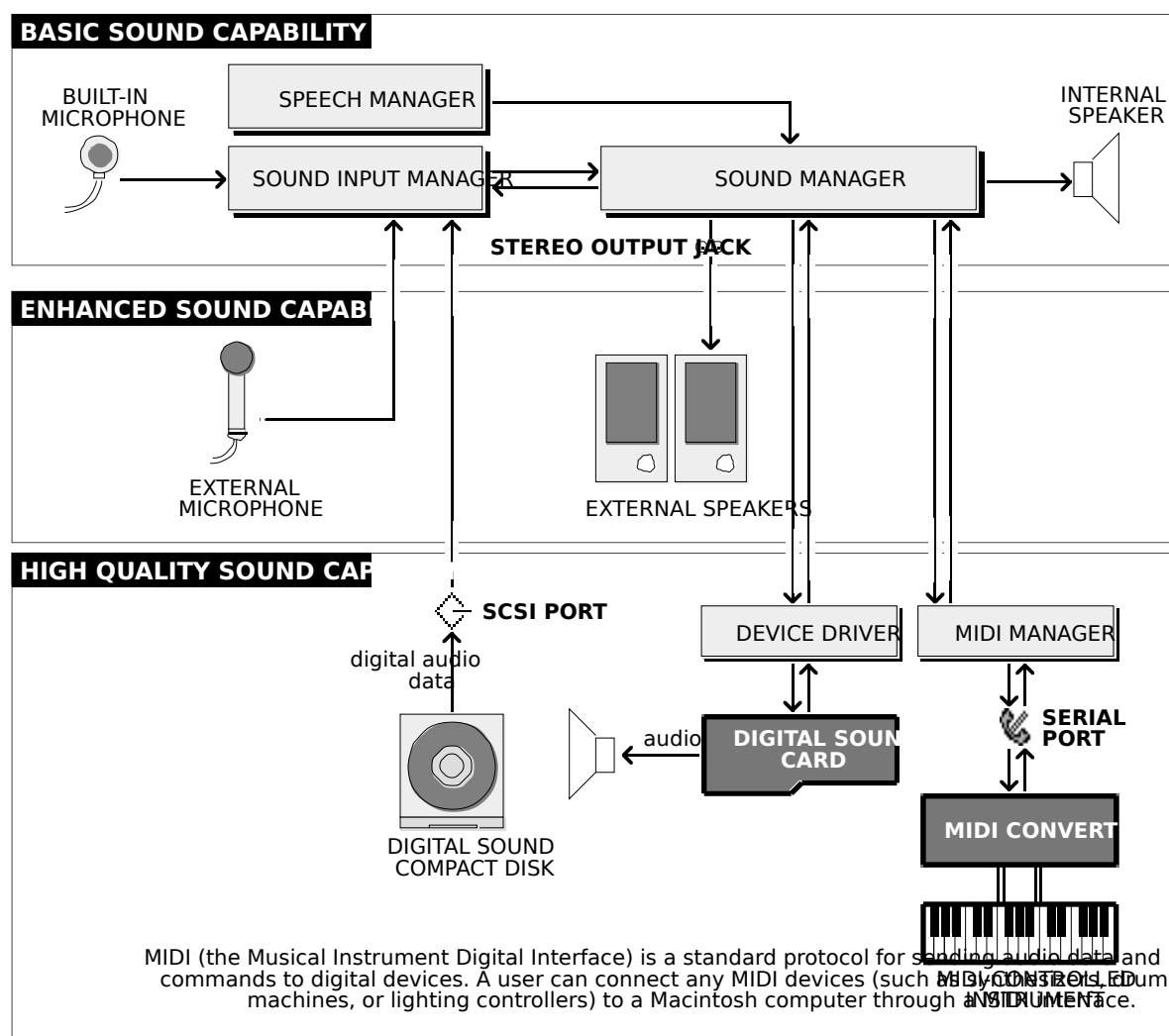


FIG 1 - SOUND CAPABILITIES OF MACINTOSH COMPUTERS

Sound Data

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

- **Square Wave Data.** Square wave data is the simplest kind sound data. Your application can use square-wave data to play a simple sequence of sounds in which each sound is described completely by three factors: frequency (or pitch); amplitude (or volume); duration.
- **Wave-Table Data.** To produce more complex sounds than are possible using square-wave data, your application can use wave-table data. Wave-table data is based on a description of a single wave cycle. The wave cycle is represented as an array of 512 bytes that describe the timbre (or tone) of a sound at any point in the cycle.
- **Sampled-Sound Data.** You can use sampled-sound data to play back sounds that have been digitally recorded (that is, **sampled sounds**). Sampled sounds are a continuous list of relative voltages over time that allow the Sound Manager to reconstruct an arbitrary analog wave form. They are typically used to play back prerecorded sounds such as speech or special sound effects.

This chapter is oriented primarily towards the recording and playback of sampled sounds.

About Sampled Sound

Two basic characteristics affect the quality of sampled sound. Those characteristics are **sample rate** and **sample size**.

Sample Rate

Sample rate, or the rate at which voltage samples are taken, determines the highest possible **frequency** that can be recorded. Specifically, for a given sample rate, sounds can be sampled up to half that frequency. For example, if the sample rate is 22,254 samples per second (that is, 22,254 hertz, or Hz), the highest frequency that can be recorded is about 11,000 Hz. A commercial compact disc is sampled at 44,100 Hz, providing a frequency response of up to about 20,000 Hz, which is the limit of human hearing.

Sample Size

Sample size, or quantisation, determines the **dynamic range** of the recording (the difference between the quietest and the loudest sound). If the sample size is eight bits, 256 discrete voltage levels can be recorded. This provides approximately 48 decibels (dB) of dynamic range. A compact disc's sample size is 16 bits, which provides about 96 dB of dynamic range. (Humans with good hearing are sensitive to ranges greater than 100 dB.)

Sound Manager Capabilities

The current Sound Manager supports 16-bit stereo audio samples with sample rates up to 64kHz, which allows your application to produce CD-quality sound. On Macintosh models which do not have the hardware to output 16-bit sound, the Sound Manager automatically converts 16-bit samples to 8-bit samples.

Storing Sampled Sounds

Sampled-sound data is made up of a series of **sample frames**, which are stored contiguously in order of increasing time. You can use the Sound Manager to store sampled sounds in one of two ways, either in **sound resources** or in **sound files**.

Sound Components

The Sound Manager supports arbitrary modifications of 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 and for converting sample rates. Sound components may be hooked together in series to perform complex tasks, as shown in the example at Fig 2.

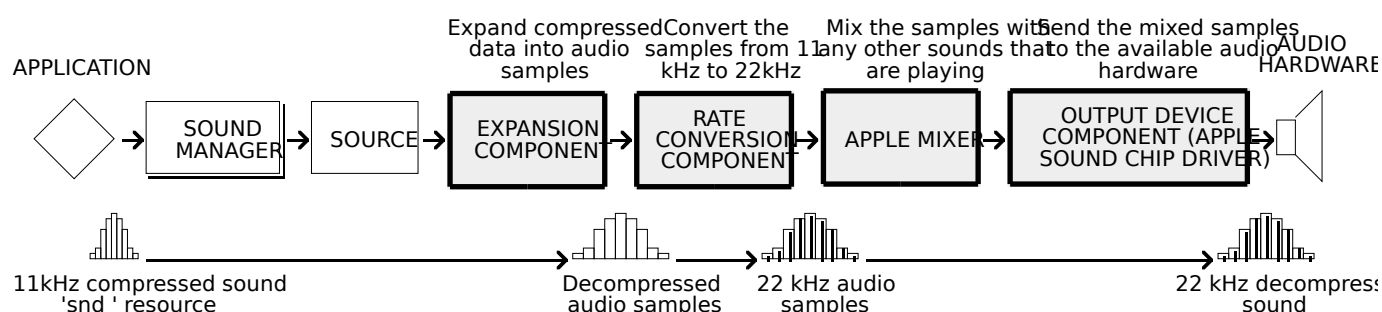


FIG 2 - A TYPICAL SOUND COMPONENT CHAIN

Compression/Decompression Components. Components which compress and decompress sound are called **codecs** (compression/decompression components). Apple Computer supplies codecs that can handle 3:1 and 6:1 compression and expansion, which are suitable for most audio requirements. The Sound Manager can use any available codec to handle compression and expansion of audio data.¹

¹A term closely associated with the subject of codecs is **MACE (Macintosh Audio Compression and Expansion)**. MACE is a collection of Sound Manager functions which provide audio data compression and expansion capabilities in ratios of either 3:1 or 6:1. The Sound Manager uses codecs to handle the MACE capabilities.

In general, your application is unaware of the sound component chain required to produce a 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. Accordingly, even though the capabilities of the available sound output hardware can vary greatly from one Macintosh 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 available on a particular machine.

Sound Resources and Sound Files

Sound Resources

A sound resource is a resource of type 'snd' that contains **sound commands** (see below) and possibly also **sound data**. Sound resources are widely used by Macintosh applications that produce sound and provide a simple and portable way for you to incorporate sounds into your application.

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. Some reasons for using sound files rather than sound resources are as follows:

- You want your application to play a sampled sound created by another application, or you want other applications to be able to play a sampled sound created by your application. (It is usually easier for different applications to share files than it is to share 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.² If the sound occupies more than about a half megabyte of space, you should probably store it as a file.

Sound File Formats. 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 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.³

The Sound Manager includes **play-from-disk** functions that allow you to play AIFF and AIFF-C files continuously from disk even while other tasks are executing.

Sound Production

Sound Channels

A Macintosh produces sound when the Sound Manager sends some data through a **sound channel** to the available audio hardware. A sound channel is a queue of **sound commands** (see below), together with other information about the sounds to be played

²Resources are limited in size by the structure of resource files and, in particular, because offsets to resource data are stored as 24-bit quantities.

³Do not confuse AIFF and AIFF-C files (referred to in this chapter as sound files) with **Finder sound files**. A Finder sound file contains a sound resource that plays when the user double clicks on the file in the Finder. 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 functions to open the Finder sound file and then by using the SndPlay function to play the single sound resource contained in it.

in that channel. The commands placed into the channel might originate from an application or from the Sound Manager itself.

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

```
type
SndChannel = PACKED RECORD
  nextChan:      SndChannelPtr;      { Pointer to next channel. }
  firstMod:      Ptr;                { (Used internally.) }
  callBack:      SndCallBackUPP;     { Pointer to callback function. }
  userInfo:      LONGINT;            { Free for application's use. }
  wait:          LONGINT;            { (Used internally.) }
  cmdInProgress: SndCommand;         { (Used internally.) }
  flags:         INTEGER;            { (Used internally.) }
  qLength:       INTEGER;            { (Used internally.) }
  qHead:         INTEGER;            { (Used internally.) }
  qTail:         INTEGER;            { (Used internally.) }
  queue:         ARRAY [0..127] OF SndCommand; { (Used internally.) }
END;

SndChannelPtr = ^SndChannel;
```

Multiple Sound Channels

It is possible to have several channels of sound open at one time. The Sound Manager (using the Apple Mixer sound 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.

Sound Commands

When you call the appropriate Sound Manager function to play a sound, the Sound Manager 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. The structure of a sound command is defined by the `SndCommand` data type:

```
type
SndCommand = PACKED RECORD
  cmd:           UInt16;              { Command number. }
  param1:        INTEGER;            { First parameter. }
  param2:        LONGINT;            { Second parameter }
END;

SndCommandPtr = ^SndCommand;
```

The Sound Manager provides a rich set of sound commands, which are defined by constants. Some examples are as follows:

<code>quietCmd</code>	= 3	Stop the sound currently playing.
<code>flushCmd</code>	= 4	Remove all commands currently queued in specified sound channel.
<code>syncCmd</code>	= 14	Synchronise multiple channels of sound.
<code>freqCmd</code>	= 42	Change the frequency of the sound. If the sound is not currently playing, begin playing at the frequency specified in param2.
<code>ampCmd</code>	= 43	Change the amplitude of the sound.
<code>soundCmd</code>	= 80	Install a sampled sound as a voice in a channel.
<code>bufferCmd</code>	= 81	Play a buffer of sampled-sound data.
<code>rateCmd</code>	= 82	Set the pitch of a sampled sound.

Sound Commands In 'snd' Resources

A simple way to issue sound commands is to call the function `SndPlay`, specifying a sound resource of type 'snd' that contains the sound commands you want to issue. A sound resource can contain any number of sound commands. As a result, you might be able to satisfy your sound-related requirements simply by creating sound resources and calling `SndPlay`.

Often, a 'snd' resource consists only of a single sound command (usually the `bufferCmd` command) together with data that describes a sampled sound to be played. The following is an example of such a 'snd' resource, shown in the form of the output of the MPW tool `DeRez` when applied to the resource:

```
data 'snd ' (19068,"Looped sound",purgeable)
{
  /* Sound resource header */
  $"0001"      /* Format type. */
  $"0001"      /* Number of data types. */
  $"0005"      /* Sampled-sound data. */
  $"00000080"  /* Initialisation option: initMono. */
  /* Sound commands */
  $"0001"      /* Number of sound commands that follow (1). */
  $"8051"      /* Command 1 (bufferCmd). */
  $"0000"      /* param1 = 0. */
  $"00000014"  /* param2 = offset to sound header (20 bytes). */
  /* Sampled sound header (Standard sound header)*/
  $"00000000"  /* samplePtr      Pointer to data (it follows immediately). */
  $"00000BB8"  /* length          Number of bytes in sample (3000 bytes). */
  $"56EE8BA3"  /* sampleRate       Sampling rate of this sound (22 kHz). */
  $"000007D0"  /* loopStart        Starting of the sample's loop point. */
  $"00000898"  /* loopEnd          Ending of the sample's loop point. */
  $"00"        /* encode           Standard sample encoding. */
  $"3C"        /* baseFrequency    BaseFrequency at which sample was taken. */
               /* sampleArea[]    Sampled sound data */
  $"80 80 81 81 81 81 81 81 80 80 80 80 81 82 82"
  $"82 83 82 82 81 80 80 7F 7F 7F 7E 7D 7D 7D 7C 7C"
  (Rest of sampled sound data.)
};
```

This resource indicates that the sound is defined using sampled-sound data and includes a call to a single sound command (the `bufferCmd` command). The offset bit of the command number is set to indicate that the sound data is contained within the resource itself. (Data can also be stored in a buffer separate from a sound resource.) The second parameter to the `bufferCmd` command indicates the offset from the beginning of the resource to the **sampled sound header**⁴, which immediately follows the command and its two parameters. Note that the first part of the sampled sound header contains important information about the sample and that the sampled sound data is itself part of the sampled sound header. Note also the `loopStart` and `loopEnd` fields of the sampled sound header, which are central to the matter of looping a sound indefinitely.

Sending Sound Commands Directly From the Application

You can also send sound commands one at a time into a sound channel by repeatedly calling the `SndDoCommand` function. The commands are held in a queue and processed in a first-in, first-out order. Alternatively, you can bypass a sound queue altogether by calling the `SndDoImmediate` function

Synchronous and Asynchronous Sound

You can play sounds either **synchronously** or **asynchronously**.

Synchronous Sound

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.

⁴The sampled sound header shown is a **standard sound header**, which can reference only buffers of monophonic 8-bit sound. The **extended sound header** is used for 8-bit or 16-bit stereo sound data as well as monophonic sound data. The **compressed sound header** is used to describe compressed sound data, whether monophonic or stereo.

Asynchronous Sound

When you play a sound asynchronously, your application can continue other processing while the sound is playing. From a programming standpoint, asynchronous sound production is considerably more complex than synchronous sound production.

Playing a Sound

Playing a Sound Resource

You can load a sound resource into memory and then play it using the `SndPlay` function. As previously stated, a 'snd' resource contains sound commands that play the desired sound and might also contain sound data. If it does contain sound data, that data might be either compressed or noncompressed. `SndPlay` decompresses the data, if necessary, to play the sound.

Channel Allocation. When you pass `SndPlay` a NULL sound channel pointer in its first parameter, the Sound Manager automatically allocates a sound channel for the sound and then disposes of the channel when the sound has completed playing. The sound channel is allocated in the application's heap.

Playing a Sound File

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.

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 minimises RAM usage (at the expense of additional disk overhead). `SndStartFilePlay` is thus ideal for playing very large sounds.

Channel Allocation. When you pass `SndStartFilePlay` a NULL sound channel pointer in the first parameter, the Sound Manager automatically allocates a sound channel for the sound.

Checking For Play-From-Disk Capability. The Sound Manager supports play-from-disk only on certain Macintosh computers. Accordingly, you should use the `Gestalt` function (see Chapter 23 — Miscellany) to check for this capability before calling `SndStartFilePlay`.

Playing Sounds Asynchronously

The Sound Manager allows you to play sounds asynchronously only if you allocate sound channels yourself. If you use such a technique, your application will need to dispose of a sound channel whenever the application finishes playing a sound. In addition, your application might need to release a sound resource that you played on a sound channel.

The Sound Manager provides certain mechanisms that allow your application to ascertain when a sound finishes playing, so that it can arrange to dispose of, firstly, a sound channel no longer being used and, secondly, other data (such as a sound resource) that you no longer need after disposing of the channel. Despite the existence of these mechanisms, the programming aspects of asynchronous sound remain rather complex. For that reason, the demonstration program files associated with this chapter include a library, called `AsynchSoundLib` (`AsynchSoundLib68K` for the 680x0 version or `AsynchSoundLibPPC` for the PowerPC version), which support asynchronous sound playback and which eliminates the necessity for your application to itself include source code relating to the more complex aspects of asynchronous sound management.

AsynchSoundLib, which may be used by any application that requires a straightforward and uncomplicated interface for asynchronous sound playback, is documented following the Constants, Data Types, and Functions section of this chapter.

Sound Recording

The Sound Input Manager provides the ability to record and digitally store sounds in a device-independent manner, and provides two high-level functions that allow your application to record sounds from the user and store them in memory or in a file. When you call these functions, the Sound Input Manager presents the sound recording dialog box shown at Fig 3.

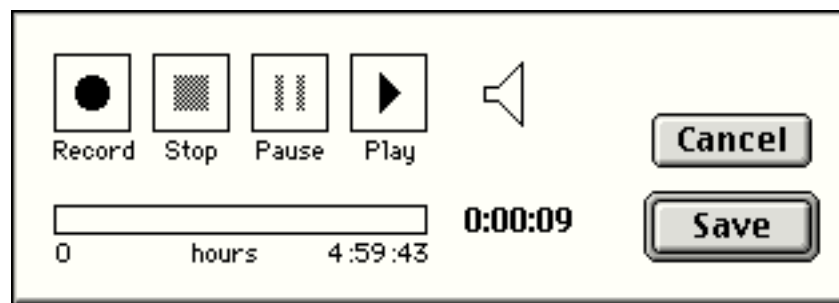


FIG 3 - SOUND RECORDING DIALOG

Recording a Sound Resource

You can record sounds from the current input device using the `SndRecord` function. When calling `SndRecord`, you can pass a handle to a block of memory as the fourth parameter. The incoming data will then be stored in that block, the size of which determines the recording time available. If you pass `NULL` as the fourth parameter, the Sound Input Manager allocates the largest possible block in the application heap. Either way, the Sound Input Manager resizes the block when the user clicks the `Save` button.

When you have recorded a sound, you can play it back by calling `SndPlay` and passing it the handle to the block of memory in which the sound data is stored. That block has the *structure* of a 'snd' resource, but its handle is not a handle to an existing resource. To save the recorded data as a resource, you can use the appropriate Resource Manager functions in the usual way.

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 a block of 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.

Recording Quality

One of the following constants should be passed in the third parameter of both the `SndRecord` and the `SndRecordToFile` call so as to specify the recording quality required:

Constant	Value	Meaning
siCDQuality	'cd '	44.1kHz, stereo, 16 bit.
siBestQuality	'best'	22kHz, mono, 8 bit.
siBetterQuality	'betr'	22kHz, mono, 3:1 compression.
siGoodQuality	'good'	22KHz, mono, 6:1 compression

The highest quality sound naturally requires the greatest storage space. Accordingly, be aware that, for most voice recording, you should specify `siGoodQuality`.

As an example of the storage space required for sounds, one minute of monophonic sound recorded with the fidelity you would expect from a commercial compact disc occupies about 5.3 MB of disk space. Even one minute of telephone-quality speech takes up more than half a megabyte.

Checking For Sound Recording Equipment

Not all Macintosh models support sound recording. Accordingly, before calling `SndRecord` or `SndRecordToFile`, you must use the `Gestalt` function to determine whether sound-recording hardware and software are installed.

Speech

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 do not need to be familiar with the Sound Manager to take advantage of the Speech Manager's capabilities.

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 synthesiser**, a component that contains executable code that manages all communication between the Speech Manager and the Sound Manager. A synthesiser is usually contained in a resource in a file within the System folder. A speech synthesiser can include one or more voices, each of which may have different tonal qualities.

Generating Speech From a String

The `SpeakString` function is used to convert a text string into speech. `SpeakString` automatically allocates a speech channel, uses that channel to produce speech, and then disposes of the speech channel.

Asynchronous Speech

Speech generation is asynchronous, that is, control returns to your application before `SpeakString` finishes speaking the string. However, 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.

Synchronous Speech

If you wish 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.

Checking For Speech Capabilities

Because the Speech Manager is not available in all system software versions, your application should always check for speech capabilities, using the Gestalt function, before calling `SpeakString` or `SpeechBusy`.

Relevant Constants, Data Types, and Functions

Constants

Gestalt Sound Attributes Selector and Response Bits

<code>gestaltSoundAttr</code>	<code>'snd'</code>	Sound attributes.
<code>gestaltStereoCapability</code>	<code>= 0</code>	Sound hardware has stereo capability.
<code>gestaltStereoMixing</code>	<code>= 1</code>	Stereo mixing on external speaker.
<code>gestaltSoundIOMgrPresent</code>	<code>= 3</code>	Sound I/O Manager is present.
<code>gestaltBuiltinSoundInput</code>	<code>= 4</code>	Built-in Sound Input hardware is present.
<code>gestaltHasSoundInputDevice</code>	<code>= 5</code>	Sound Input device available.
<code>gestaltPlayAndRecord</code>	<code>= 6</code>	Built-in hardware can play & record simultaneously.
<code>gestalt16BitSoundIO</code>	<code>= 7</code>	Sound hardware can play and record 16-bit samples.
<code>gestaltStereoInput</code>	<code>= 8</code>	Sound hardware can record stereo.
<code>gestaltLineLevelInput</code>	<code>= 9</code>	Sound input port requires line level.
<code>gestaltSndPlayDoubleBuffer</code>	<code>= 10</code>	<code>SndPlayDoubleBuffer</code> available.
<code>gestaltMultiChannels</code>	<code>= 11</code>	Multiple channel support.
<code>gestalt16BitAudioSupport</code>	<code>= 12</code>	16 bit audio data supported.
<code>gestaltSpeechAttr</code>	<code>'ttsc'</code>	Speech Manager attributes.
<code>gestaltSpeechMgrPresent</code>	<code>= 0</code>	Speech Manager exists.
<code>gestaltSpeechHasPPCglue</code>	<code>= 1</code>	Native PPC glue for Speech Manager API exists.

Recording Qualities

<code>siCDQuality</code>	<code>= 'cd'</code>	44.1kHz, stereo, 16 bit.
<code>siBestQuality</code>	<code>= 'best'</code>	22kHz, mono, 8 bit.
<code>siBetterQuality</code>	<code>= 'betr'</code>	22kHz, mono, MACE 3:1.
<code>siGoodQuality</code>	<code>= 'good'</code>	22kHz, mono, MACE 6:1.

Typical Sound Commands

<code>quietCmd</code>	<code>= 3</code>	Stop the sound currently playing.
<code>flushCmd</code>	<code>= 4</code>	Remove all commands currently queued in the specified sound channel.
<code>syncCmd</code>	<code>= 14</code>	Synchronise multiple channels of sound.
<code>freqCmd</code>	<code>= 42</code>	Change the frequency of the sound. If the sound is not currently playing, begin playing indefinitely at the frequency specified in param2.
<code>ampCmd</code>	<code>= 43</code>	Change the amplitude of the sound.
<code>soundCmd</code>	<code>= 80</code>	Install a sampled sound as a voice in a channel.
<code>bufferCmd</code>	<code>= 81</code>	Play a buffer of sampled-sound data.
<code>rateCmd</code>	<code>= 82</code>	Set the pitch of a sampled sound.

Data Types

Sound Channel Structure

```
SndChannel = PACKED RECORD
  nextChan:      SndChannelPtr;      { Pointer to next channel. }
  firstMod:      Ptr;                { (Used internally.) }
  callBack:      SndCallBackUPP;     { Pointer to callback function. }
  userInfo:      LONGINT;            { Free for application's use. }
  wait:          LONGINT;            { (Used internally.) }
  cmdInProgress: SndCommand;         { (Used internally.) }
  flags:         INTEGER;            { (Used internally.) }
  qLength:       INTEGER;            { (Used internally.) }
  qHead:         INTEGER;            { (Used internally.) }
  qTail:         INTEGER;            { (Used internally.) }
  queue:         ARRAY [0..127] OF SndCommand; { (Used internally.) }
END;
```

```
SndChannelPtr = ^SndChannel;
```

Sound Command Structure

```
SndCommand = PACKED RECORD
  cmd:      Uint16;      { Command number. }
  param1:   INTEGER;    { First parameter. }
  param2:   LONGINT;    { Second parameter }
  END;
```

```
SndCommandPtr = ^SndCommand;
```

Functions

Playing Sound Resources

```
PROCEDURE SysBeep(duration: INTEGER);
FUNCTION SndPlay(chan: SndChannelPtr; sndHandle: SndListHandle; async: BOOLEAN): OSErr;
```

Playing From Disk

```
FUNCTION SndStartFilePlay(chan: SndChannelPtr; fRefNum: INTEGER; resNum: INTEGER;
  bufferSize: LONGINT; theBuffer: UNIV Ptr; theSelection: AudioSelectionPtr;
  theCompletion: FilePlayCompletionUPP; async: BOOLEAN): OSErr;
FUNCTION SndPauseFilePlay(chan: SndChannelPtr): OSErr;
FUNCTION SndStopFilePlay(chan: SndChannelPtr; quietNow: BOOLEAN): OSErr;
```

Allocating and Releasing Sound Channels

```
FUNCTION SndNewChannel(VAR chan: SndChannelPtr; synth: INTEGER; init: LONGINT;
  userRoutine: SndCallBackUPP): OSErr;
FUNCTION SndDisposeChannel(chan: SndChannelPtr; quietNow: BOOLEAN): OSErr;
```

Sending Commands to a Sound Channel

```
FUNCTION SndDoCommand(chan: SndChannelPtr; {CONST}VAR cmd: SndCommand;
  noWait: BOOLEAN): OSErr;
FUNCTION SndDoImmediate(chan: SndChannelPtr; {CONST}VAR cmd: SndCommand): OSErr;
```

Recording Sounds

```
FUNCTION SndRecord(filterProc: ModalFilterUPP; corner: Point; quality: OSType;
  VAR sndHandle: SndListHandle): OSErr;
FUNCTION SndRecordToFile(filterProc: ModalFilterUPP; corner: Point; quality: OSType;
  fRefNum: INTEGER): OSErr;
```

Generating Speech

```
FUNCTION SpeakString(textToBeSpoken: Str255): OSErr;
FUNCTION SpeechBusy: INTEGER;
```

The AsynchSoundLib Library

The AsynchSoundLib library is intended to provide a straightforward and uncomplicated interface for asynchronous sound playback.

AsynchSoundLib requires that you include a global "attention" flag in your application. At startup, your application must call AsynchSoundLib's initialisation function and provide the address of this attention flag. Thereafter, the application must continually check the attention flag within its main event loop.

AsynchSoundLib's main function is to spawn asynchronous sound tasks, and communication between your application and AsynchSoundLib is carried out on an as-required basis. The basic phases of communication for a typical sound playback sequence are as follows.

- Your application tells AsynchSoundLib to play some sound.

- `AsynchSoundLib` uses the Sound Manager to allocate a sound channel and begins asynchronous playback of your sound.
- The application continues executing, with the sound playing asynchronously in the background.
- The sound completes playback. `AsynchSoundLib` has set up a sound command that causes it (`AsynchSoundLib`) to be informed immediately upon completion of playback. When playback ceases, `AsynchSoundLib` sets the application's global attention flag.
- The next time through your application's event loop, the application notices that the attention flag is set and calls `AsynchSoundLib` to free up the sound channel.

When your application terminates, it must call `AsynchSoundLib` to stop any asynchronous playback in progress at the time.

`AsynchSoundLib`'s method of communication with the application minimises processing overhead. By using the attention flag scheme, your application calls `AsynchSoundLib`'s cleanup function only when it is really necessary.

AsynchSoundLib Functions

The following documents those `AsynchSoundLib` functions that may be called from an application.

To facilitate an understanding of the following, it is necessary to be aware that `AsynchSoundLib` associates a data structure, referred to in the following as an **ASStructure**, with each channel. Each `ASStructure` includes the following fields:

<code>channel : SndChannel;</code>	{ The sound channel. }
<code>refNum : SInt32;</code>	{ Reference number. }
<code>sound : Handle; { The sound. }</code>	
<code>handleState : UInt8;</code>	{ State to which to restore the sound handle. }
<code>inUse : Boolean;</code>	{ Is this <code>ASStructure</code> currently in use? }

function **AS_Initialise** (*attnFlag,numChannels*) : `OSErr`;

<code>var attnFlag : Boolean;</code>	Application's "attention" flag global variable.
<code>numChannels : SInt16;</code>	Number of channels required to be open simultaneously. If 0 is specified, <code>numChannels</code> defaults to 4.

Returns: 0 No errors.
Non-zero results of `MemError` call.

This function stores the address of the application's "attention" flag global variable and then allocates memory for a number of `ASStructures` equal to the requested number of sound channels.

function **AS_PlayID** (*resID,refNum*) : `OSErr`;

<code>resID : SInt16;</code>	Resource ID of the 'snd ' resource.
<code>var refNum : SInt32;</code>	A pointer to a reference number storage variable. Optional.

Returns: 0 No errors.
1 No channels available.
Non-zero results of `ResError` call.
Non-zero results of `SndNewChannel` call.
Non-zero results of `SndPlay` call.

This function initiates asynchronous playback of the 'snd ' resource with ID `resID`.

Note: If you pass a pointer to a variable in their `refNum` parameters, `AS_PlayID` and its sister function `AS_PlayHandle` (see below) return a reference number in that parameter. As will be seen, this reference number may be used to gain more control over the

playback process. However, if you simply want to trigger a sound and let it to run to completion, with no further control over the playback process, you can pass NULL in the `refNum` parameter. In this case, a reference number will not be returned.

First, `AS_PlayID` attempts to load the specified 'snd ' resource. If successful, the handle state is saved for later restoration, and the handle is made un purgeable. The function then gets a reference number and a pointer to the next free `ASStructure`. A sound channel is then allocated via a call to `SndNewChannel` and the associated `ASStructure` is initialised. `HLockHi` is then called to move the sound handle high in the heap and lock it. `SndPlay` is then called to start the sound playing, playing, the `channel.userInfo` field is set to indicate that the sound is playing, and a callback function is queued so that `AsynchSoundLib` will know when the sound has stopped playing. If all this is successful, `AS_PlayID` returns the reference number associated with the channel (if the caller wants it).

```
function AS_PlayHandle(sound,refNum) : OSErr;
```

```
sound : Handle;           A handle to the sound to be played.
var refNum : SInt32;      A pointer to a reference number storage variable. Optional.
```

Returns: 0 If no errors.
 1 No channels available.
 Non-zero results of `SndNewChannel` call.
 Non-zero results of `SndPlay` call.

This function initiates asynchronous playback of the sound referred to by `sound`.

Note: The `AS_PlayHandle` function is similar to `AS_PlayID`, except that it supports a special case: You can pass `AS_PlayHandle` a nil handle. This causes `AS_PlayHandle` to open a sound channel but not call `SndPlay`. Normally, you do this when you want to get a sound channel and then send sound commands directly to that channel yourself. (See `AS_GetChannel`, below.)

If a handle is provided, its current state is saved for later restoration before it is made un purgeable. `AS_PlayHandle` then gets a reference number and a pointer to a free `ASStructure`. A sound channel is then allocated via a call to `SndNewChannel` and the associated `ASStructure` is initialised. Then, if a handle was provided, `HLockHi` is called to move the sound handle high in the heap and lock it, following which `SndPlay` is called to start the sound playing, the `channel.userInfo` field is set to indicate that the sound is playing, and a callback function is queued so that `AsynchSoundLib` will know when the sound has stopped playing. Finally, the reference number associated with the channel is returned (if the caller wants it).

```
function AS_GetChannel(refNum,channel) : OSErr;
```

```
refNum : SInt32           Reference number.
var channel : SndChannelPtr A pointer to a SoundChannelPtr.
```

Returns: 0 No errors.
 2 If `refNum` does not refer to any current `ASStructure`.

This function searches for the `ASStructure` associated with `refNum`. If one is found, a pointer to the associated sound channel is returned in the `channel` parameter.

`AS_GetChannel` is provided so as to allow an application to gain access to the sound channel associated with a specified reference number and thus gain the potential for more control over the playback process. It allows an application to use `AsynchSoundLib` to handle sound channel management while at the same time retaining the ability to send sound commands to the channel. This is most commonly done to play looped continuous music, for which you will need to provide a sound resource with a loop and a sound command to install the music as a voice. First, you open a channel by calling `AS_PlayHandle`, specifying nil in the first parameter. (This causes `AS_PlayHandle` to open a sound channel but not call `SndPlay`.) Armed with the returned reference number associated with that channel, you then call `AS_GetChannel` to get the `SndChannelPtr`, which you then pass as the first parameter in a call to `SndPlay`. Finally, you send a `freqCmd` command to the channel to start the music playing. The playback will keep looping until you send a `quietCmd` command to the channel.

procedure **AS_CloseChannel**;

This function is called from the application's event loop if the application's "attention" flag is set. It clears the "attention" flag and then performs playback cleanup by iterating through the ASStructures looking for structures which are both in use (that is, the inUse field contains true) *and* complete (that is, the channel.userInfo field has been set by AsyncSoundLib's callback function to indicate that the sound has stopped playing). It frees up such structures for later use and closes the associated sound channel.

procedure **AS_CloseDown**;

AS_CloseDown checks that AsyncSoundLib was previously initialised, stops all current playback, calls AS_CloseChannel to close open sound channels, and disposes of the associated ASStructures.

Demonstration Program

```
{
// SoundDemo.p
//
// This program opens a modal dialog containing eight bevel button controls arranged in
// two groups, namely, a synchronous sound group and an asynchronous sound group.
// Clicking on the bevel buttons causes sound to be played back or recorded as follows:
//
// • Synchronous group:
//   • Play sound resource.
//   • Play sound file.
//   • Record sound resource.
//   • Record sound file.
//   • Speak text string.
//
// • Asynchronous group:
//   • Start and stop looped sound playback.
//   • Play unlooped sound.
//   • Speak text string.
//
// At startup, the program checks for play-from-disk, sound recording capability, speech
// capability, and multi-channel capability. If these are not available, the relevant
// buttons are disabled.
//
// The asynchronous sound sections of the program utilise a special library called
// AsyncSoundLib, which must be included in the CodeWarrior project (AsyncSoundLib68K for
// 68$0 projects and AsyncSoundLibPPC for PowerPC projects).
//
// The program utilises the following resources:
//
// • 'MBAR' and 'MENU' resources (preload, non-purgeable).
//
// • A 'DLOG' resource and associated 'DITL', 'dlgx', and 'dftb' resources (all
// purgeable).
//
// • 'CNTL' resources (purgeable) for the controls within the dialog.
//
// • Three 'snd ' resources, one for synchronous playback (purgeable), one for looped
// asynchronous playback (unpurgeable), and one for unlooped asynchronous playback
// (purgeable).
//
// • Four 'cicn' resources (purgeable). Two are used to provide an animated display
// which halts during synchronous playback and continues during asynchronous playback.
// The remaining two are used by the bevel button controls.
//
// • Three 'STR#' resources containing error message strings and 'speak text' strings
// (all purgeable).
```




```

        end;

    if (sndListHdl <> nil) then
        begin
            HLock(Handle(sndListHdl));
            osErr := SndPlay(nil, sndListHdl, false);
            if (osErr <> noErr) then
                begin
                    DoErrorAlertWithCode(eSndPlay, osErr);
                end;
            HUnlock(Handle(sndListHdl));
            ReleaseResource(Handle(sndListHdl));

            ignoredErr := GetDialogItemAsControl(gDialogPtr, iPlayResource, controlHdl);
            SetControlValue(controlHdl, 0);
        end;
    end;
    { of procedure DoPlayResource }

//  DoPlayFile

procedure DoPlayFile;
    var
        osErr : OSErr;
        fileSysSpec : FSSpec;
        fileRefNum : SInt16;
        controlHdl : ControlHandle;
        ignoredErr : OSErr;

    begin
        osErr := FSMakeFSSpec(0, 0, ':soundfile.aiff', fileSysSpec);
        if (osErr = noErr) then
            begin
                osErr := FSpOpenDF(fileSysSpec, fsRdPerm, fileRefNum);
            end;


            if (osErr = noErr) then
                begin
                    ignoredErr := SetFPos(fileRefNum, fsFromStart, 0);
                end;

                if (osErr = noErr) then
                    begin
                        osErr := SndStartFilePlay(nil, fileRefNum, 0, 20480, nil, nil, nil, false);
                    end;

                    if (osErr <> noErr) then
                        begin
                            DoErrorAlertWithCode(ePlayFile, osErr);
                        end;

                        ignoredErr := FSClose(fileRefNum);

                        ignoredErr := GetDialogItemAsControl(gDialogPtr, iPlayFile, controlHdl);
                        SetControlValue(controlHdl, 0);
                    end;
                end;
            { of procedure DoPlayFile }

//  DoRecordResource

procedure DoRecordResource;
    var
        oldResFileRefNum : SInt16;
        topLeft : Point;
        soundHdl : Handle;
        osErr, memErr : OSErr;
        theResourceID, resErr : SInt16;
        controlHdl : ControlHandle;
        ignoredErr : OSErr;

    begin
        oldResFileRefNum := CurResFile;
        UseResFile(gAppResFileRefNum);

        topLeft.h := (qd.screenBits.bounds.right div 2) - 156;
        topLeft.v := 150;

        soundHdl := NewHandle(25000);
        memErr := MemError;
        if (memErr <> noErr) then

```

Version 2.1

```
begin
DoErrorAlert(eMemory);
Exit(DoRecordResource);
end;

osErr := SndRecord(nil, topLeft, siBetterQuality, SndListHandle(soundHdl));
if ((osErr <> noErr) and (osErr <> userCanceledErr)) then
begin
DoErrorAlertWithCode(eSndRecord, osErr);
end
else begin
repeat
begin
theResourceID := UniqueID('snd ');
end;
until (theResourceID > 8191);

AddResource(Handle(soundHdl), 'snd ', theResourceID, 'Test');
resErr := ResError;
if (resErr = noErr) then
begin
UpdateResFile(gAppResFileRefNum);
end;

resErr := ResError;
if (resErr <> noErr) then
begin
DoErrorAlertWithCode(eWriteResource, resErr);
end;
end;

UseResFile(oldResFileRefNum);

ignoredErr := GetDialogItemAsControl(gDialogPtr, iRecordResource, controlHdl);
SetControlValue(controlHdl, 0);
end;
{ of procedure DoRecordResource }
```

```
// DoRecordFile
procedure DoRecordFile;
var
topLeft : Point;
osErr : OSErr;
fileSysSpec : FSSpec;
fileRefNum : SInt16;
controlHdl : ControlHandle;
ignoredErr : OSErr;

begin
topLeft.h := (qd.screenBits.bounds.right div 2) - 156;
topLeft.v := 150;

osErr := FSMakeFSSpec(0, 0, ':test.aiff', fileSysSpec);
if (osErr = fnfErr) then
begin
osErr := FSpCreate(fileSysSpec, '????', 'AIFF', smSystemScript);
end;

if (osErr = noErr) then
begin
osErr := FSpOpenDF(fileSysSpec, fsWrPerm, fileRefNum);
end;

if (osErr = noErr) then
begin
ignoredErr := SetFPos(fileRefNum, fsFromStart, 0);
end;

if (osErr = noErr) then
begin
osErr := SndRecordToFile(nil, topLeft, siBetterQuality, fileRefNum);
end;

if ((osErr <> noErr) and (osErr <> userCanceledErr)) then
begin
DoErrorAlertWithCode(eRecordFile, osErr);
end;

ignoredErr := FSClose(fileRefNum);
```



```

    ignoredErr := GetDialogItemAsControl(gDialogPtr, iRecordFile, controlHdl);
    ignoredErr := DeactivateControl(controlHdl);
end;

if not gHasSpeechManager then
begin
    ignoredErr := GetDialogItemAsControl(gDialogPtr, iSpeakTextSync, controlHdl);
    ignoredErr := DeactivateControl(controlHdl);
    ignoredErr := GetDialogItemAsControl(gDialogPtr, iSpeakTextAsync, controlHdl);
    ignoredErr := DeactivateControl(controlHdl);
end;

if not gHasMultiChannel then
begin
    ignoredErr := GetDialogItemAsControl(gDialogPtr, iLoopedSound, controlHdl);
    ignoredErr := DeactivateControl(controlHdl);
end;
end;
{ of procedure DoSetUpDialog }

// ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦ DoAdjustItems

procedure DoAdjustItems;
var
    controlHdl : ControlHandle;
    a : SInt16;
    ignoredErr : OSErr;

begin
    ignoredErr := GetDialogItemAsControl(gDialogPtr, iLoopedSound, controlHdl);
    if gLoopedSoundOn then
        begin
            SetControlTitle(controlHdl, 'Switch Looped Sound Off');
        end
    else begin
        SetControlTitle(controlHdl, 'Switch Looped Sound On');
    end;

    for a := iRecordResource to iRecordFile do
        begin
            ignoredErr := GetDialogItemAsControl(gDialogPtr, a, controlHdl);

            if gLoopedSoundOn then
                begin
                    ignoredErr := DeactivateControl(controlHdl);
                end
            else begin
                ignoredErr := ActivateControl(controlHdl);
            end;
        end;
    end;
    { of procedure DoAdjustItems }

// ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦ DoErrorAlert

procedure DoErrorAlert(stringIndex : SInt16);
var
    errorString : Str255;
    ignoredErr : OSErr;

begin
    GetIndString(errorString, rErrorStrings, stringIndex);
    ParamText(errorString, ", ", "");
    ignoredErr := StopAlert(rErrorAlert, nil);
end;
    { of procedure DoErrorAlert }

// ♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦ DoErrorAlertWithCode

procedure DoErrorAlertWithCode(stringIndex : SInt16; resultCode : SInt16);
var
    errorString, resultCodeString : Str255;
    ignoredErr : OSErr;

begin
    GetIndString(errorString, rErrorStringsWithCode, stringIndex);
    NumToString(SInt32(resultCode), resultCodeString);


    ParamText(errorString, resultCodeString, ", ");
    ignoredErr := StopAlert(rErrorAlertWithCode, nil);
end;

```



```
//
.....
..... enter event loop

    EventLoop;
end.
    { of main program block }

// 
```

Demonstration Program Comments

Ensure that the Speech Manager extension is on before running this program.

When this program is run, the user should click on the various buttons in the dialog box to record and play back sound resources and sound files and to play back the provided "speak text" strings. The user should observe the effects of asynchronous and synchronous playback on the "working man" icon in the image well in the dialog. The user should also observe that the text "AS_CloseChannel called" appears briefly in the secondary group box to the right of the image well when AsynchSoundLib sets the application's "attention" flag to true, thus causing the application to call the AsynchSoundLib function AS_CloseChannel.

Note that the doRecordResource function saves recorded sounds as 'snd ' resources with unique IDs in the resource fork of the application (Sound_68K or Sound_PPC). In addition, the doRecordFile function creates a file called "test.aiff" in the directory containing this application. When you have finished exploring the recording aspects of this demonstration, the you may wish to remove the file "test.aiff" and the 'snd ' resources you have created.

constants

rDialog and the following nine constants represent the dialog's resource ID and items. The next four constants represent the resource IDs of 'snd ' resources and a 'STR#' resource containing the "speak text" strings. The next eighteen constants represent error 'ALRT' resource IDs, the IDs of "STR#" resources containing error strings, and the indexes into those "STR#" resources. The next two constants represent 'cicn' resource IDs.

kMaxChannels will be used to specify the maximum number of sound channels that AsynchSoundLib is to open. kOutOfChannels will be used to determine whether the AsynchSoundLib function AS_PlayID returns a "no channels available" error.

Global Variables

The application's resource file reference number will be saved to gAppResFileRefNum at startup.

gHasSoundPlayDoubBuff, gHasSoundInputDevice, gHasSpeechmanager, and gHasMultiChannel will be set to true if the associated sound capabilities are available, otherwise they will be set to false.

gLoopedSoundOn will be toggled between true and false by successive presses of the "Switch Looped Sound On/Off" bevel button. gLoopedSoundRefNum will be assigned the reference number returned by a call to the AsynchSoundLib function AS_PlayHandle. gLoopedSoundChannel will be assigned the pointer to the sound channel structure returned by a call to the AsynchSoundLib function AS_GetChannel.

gCallAS_CloseChannel is the application's "attention" flag. This will be set to true by AsynchSoundLib when a sound played asynchronously has stopped playing.

main program block

CurResFile saves the reference number of the application's resource file. The call to doCheckSoundEnvironment checks the capability of the sound environment and sets global variables accordingly.

doInitialiseSoundLib is called to initialise the AsynchSoundLib library.

If multi-channel playback is available, the application-defined function doLoopedSoundSetup is called to set up the looped sound playback. If this call is not successful, an error alert is displayed, the AsynchSoundLib function AS_CloseDown is called and the program terminates.

This block means that, on machines without multi-channel playback capability, the program has opted to defeat the continuous looped sound playback and make the single channel available for the other playback options represented by the buttons in the dialog. The program could be readily modified to reverse this situation and allow the user to make the single channel available to the continuous looped sound only.

DoCheckSoundEnvironment

DoCheckSoundEnvironment checks for play-from-disk capability, recording capability, speech capability, and multi-channel playback capability, and sets the associated global variables accordingly.

DoInitialiseSoundLib

DoInitialiseSoundLib initialises the AsynchSoundLib library. More specifically, it calls the AsynchSoundLib function AS_Initialise and passes to AsynchSoundLib the address of the application's "attention" flag (gAS_CloseChannel), together with the requested number of channels.

If AS_Initialise returns a non-zero value, an error alert is displayed and the program terminates.

DoLoopedSoundSetUp

DoLoopedSoundSetUp gets a channel for the looped sound, loads the 'snd ' resource containing the looped sound, and calls SndPlay.

First, the AsynchSoundLib function AS_PlayHandle is called with NULL passed in the first parameter. (This causes AS_PlayHandle to open a sound channel but not call SndPlay.) The second parameter is the address of a global variable which will receive the reference number associated with the channel opened by this call to AS_PlayHandle.

If the call to AS_PlayHandle is successful, a call is made to the AsynchSoundLib function AS_GetChannel, passing the reference number returned by AS_PlayHandle in the first parameter and receiving a pointer to the sound channel in the second parameter.

If the call to AS_GetChannel is successful, GetResource attempts to load the specified 'snd ' resource. If the resource is loaded successfully, it is first moved as high in the application heap as possible and locked there. SndPlay is then called with true passed in the third parameter, indicating that asynchronous playback is required of the sound passed in the second parameter on the channel passed in the first parameter.

The 'snd ' resource being used contains one command only (soundCmd). In the standard sound header, the loopStart field contains 0 and the loopEnd field contains 24199. (The sound length is 24200 frames.) Since the soundCmd command may only be used with non-compressed sampled-sound data, the sampled sound data in the resource is not compressed.

SndPlay causes all commands and data contained in the sound handle to be sent to the channel. Since the single command in the 'snd ' resource being used is soundCmd (install a sampled sound as a voice in a channel) and not bufferCmd (play a sampled sound), nothing is heard at this point. (If the command in the resource was bufferCmd, the sound would play once at this point.)

If all four calls in DoLoopedSoundSetUp are successful, true is returned. Otherwise, false is returned and the program terminates.

EventLoop

Within the event loop, the "attention" flag required by AsynchSoundLib is checked. If AsynchSoundLib has set it to true, the AsynchSoundLib function AS_CloseChannel is called to free up the relevant ASStructure, close the relevant sound channel, and clear the "attention" flag. In addition, some text is drawn in the group box to the right of the image well to indicate to the user that AS_CloseChannel has just been called.

If WaitNextEvent retrieves an event other than a NULL event, IsDialogEvent is called to determine whether the event belongs to the dialog. If so, DialogSelect is called to determine whether one of the dialog's buttons was clicked. If so, the application-defined function doDialogHit is called to further process the item hit. If the event does not belong to the dialog, the else block supports dragging of the dialog and choosing Show/Hide Balloons from the Help menu.

If a null event was returned by WaitNextEvent, the two frames of "working man" animation are drawn within the image well, separated by five ticks, and the area in which "AS_CloseChannel called" may have been drawn is erased.

When gDone is set to true, the event loop exits, the dialog is disposed of, and the AsynchSoundLib function AS_CloseDown is called to stop all current playback, close open sound channels, and dispose of the associated ASStructures.

DoDialogHit

DoDialogHit switches according to the received item number and calls the appropriate application-defined routine to further process the item hit event.

DoPlayResource

DoPlayResource is the first of the synchronous playback functions. It uses SndPlay to play a specified 'snd ' resource.

GetResource attempts to load the resource. If the subsequent call to ResError indicates an error, an error alert is presented.

If the load was successful, the sound handle is locked prior to a call to SndPlay. Since NULL is passed in the first parameter of the SndPlay call, SndPlay automatically allocates a sound channel to play the sound and deallocates the channel when the playback is complete. false passed in the third parameter specifies that the playback is to be synchronous.

Note: The 2174-byte 'snd ' resource being used contains one command only (bufferCmd). The compressed sound header indicates MACE 3:1 compression. The loopStart field of the compressed sound header contains 6270 and the loopEnd field contains 6271. (The sound length is 6270 frames.) The 8-bit mono sound was sampled at 22kHz.

SndPlay causes all commands and data contained in the sound handle to be sent to the channel. Since there is a bufferCmd command in the 'snd ' resource, the sound is played.

If SndPlay returns an error, an error alert is presented.

When SndPlay returns, HUnlock unlocks the sound handle and ReleaseResource releases the resource.

DoPlayFile

DoPlayFile uses SndStartFilePlay to play a specified sound file.

FSMakeFSSpec converts the directory specification shown into an FSSpec structure. The pointer to the FSSpec structure returned by FSSpec is passed in the first parameter of a call to FSpOpenDF. FSpOpenDF opens the file's data fork and receives the file reference number in its third parameter. SetFPos positions the file mark to the beginning of the file.

The file reference number is passed in the second parameter of the call to SndStartFilePlay. The parameters passed to SndStartFilePlay are as follows:

- nil in the chan parameter causes SndStartPlay to allocate a sound channel itself.
- fileRefNum in the fRefNum parameter specifies the file reference number of the file to be played.
- resNum is 0 because a file is being played, not a 'snd ' resource.
- 20480 in the bufferSize parameter means the number of bytes to be allocated for input buffering.
- nil in the theBuffer parameter causes the Sound Manager to internally allocate two relocatable blocks, each of which is half the size of bufferSize.
- nil in the theSelection parameter means the entire sound will be played.
- nil in the theCompletion parameter means that there is no completion function to be called when the file has finished playing.
- false in the async parameter means that playback is to be synchronous.

If an error is detected along the way, doErrorAlert presents an error alert.

FSClose closes the file.

Note: The MACE 6:1 AIFF-C file being used was sampled at 22kHz as 8-bit mono sound. Because of the high compression, the sound quality is poor.

DoRecordResource

DoRecordResource uses SndRecord to record a sound synchronously and then saves the sound in a 'snd ' resource.

CurResFile saves the current resource file reference number and UseResFile sets the application's resource fork as the current resource file. (The 'snd ' resource will be saved to the resource fork of the application file (Sound_68K or Sound_PPC).)

The next two lines establish the location for the top left corner of the sound recording dialog.

NewHandle creates a relocatable block. The address of the handle will be passed as the fourth parameter of the SndRecord call. The size of this block determines the recording time available. (If NULL is passed as the fourth parameter of a SndRecord call, the Sound Manager allocates the largest block possible in the application's heap.) If NewHandle cannot allocate the block, an error alert is presented and the function returns.

SndRecord opens the sound recording dialog and handles all user interaction until the user clicks the Cancel or Save button. Note that the second parameter of the SndRecord call establishes the location for the top left corner of the sound recording dialog and that the third parameter specifies 22kHz, mono, 3:1 compression.

When the user clicks the Save button, the handle is resized automatically. If the user clicks the Cancel button, SndRecord returns userCanceledErr. If SndRecord returns an error other than userCanceledErr, an error alert is presented and the function returns.

The relocatable block allocated by NewHandle, and resized as appropriate by SndPlay, has the *structure* of a 'snd ' resource, but its handle is not a handle to an existing resource. To save the recorded sound as a 'snd ' resource in the application's resource fork, the do/while loop first finds an acceptable unique resource ID for the resource. (For the System

Version 2.1

file, resource IDs for 'snd ' resources in the range 0 to 8191 are reserved for use by Apple Computer, Inc. Avoiding those IDs in this demonstration is not strictly necessary, since there is no intention to move those resources to the System file.).

The call to `AddResource` causes the Resource Manager to regard the relocatable block containing the sound as a 'snd ' resource. If the call is successful, `UpdateResFile` writes the changed resource map and the 'snd ' resource to disk. If an error occurs, an error alert is presented.

`UseResFile` restores the previously saved resource file as the current resource file.

Note that, ordinarily, you should not record to your application's resource fork because applications which record to their own resource fork cannot be used over networks.

DoRecordFile

`DoRecordFile` uses `SndRecordToFile` to record a sound synchronously to a file.

The first two lines establish the location for the top left corner of the sound recording dialog.

`FSMakeFSSpec` converts the directory specification passed in its third parameter into an `FSSpec` structure. If `FSMakeFSSpec` returns `fnfErr` (file not found), `FSpCreate` creates a new file of type 'AIFF'. `FSpOpenDF` opens the file's data fork and `SetFPos` positions the file mark to the beginning of the file.

`SndRecordToFile` opens the sound recording dialog and handles all user interaction until the user clicks the Cancel or Save button. Note that the second parameter of the `SndRecord` call establishes the location for the top left corner of the sound recording dialog, that the third parameter specifies 22kHz, mono, 3:1 compression, and that the fourth parameter specifies the file reference number of the file to record to.

When `SndRecordToFile` returns, the file will contain the recorded audio data. Since compression was specified, the file will be in AIFF-C format.

If the user clicks the Cancel button, `SndRecordToFile` returns `userCanceledErr`. If an error occurs along the way and it is not `userCanceledErr`, an error alert is presented.

`FSClose` closes the file.

DoSpeakStringSync

`DoSpeakStringSync` uses `SpeakString` to speak a specified string resource and takes measures to cause the speech to be generated in a pseudo-synchronous manner.

The speech that `SpeakString` generates is asynchronous, that is, control returns to the application before `SpeakString` finishes speaking the string. In this function, `SpeechBusy` is used to cause the speech activity to be synchronous so far as the function as a whole is concerned. That is, `DoSpeakStringSync` will not return until the speech activity is complete.

As a first step, the first line saves the number of speech channels that are active immediately before the call to `SpeakString`.

`GetIndString` loads the first string from the specified 'STR#' resource. If an error occurs, an error alert is presented and the function returns.

`SpeakString`, which automatically allocates a speech channel, is called to speak the string. If `SpeakString` returns an error, an error alert is presented.

Although `SpeakString` returns control to the application immediately it starts generating the speech, the speech channel it opens remains open until the speech concludes. While the speech continues, the number of speech channels open will be one more than the number saved at the first line. Accordingly, the while loop continues until the number of open speech channels is equal to the number saved at the first line. Then, and only then, does `DoSpeakStringSync` exit.

DoLoopedSoundAsync

`DoLoopedSoundAsync` is the first of the asynchronous playback functions. It sends sound commands to the sound channel opened by the application-defined routine `DoLoopedSoundSetUp`, and on which `DoLoopedSoundSetUp` has already installed a voice.

The first line toggles the Boolean global variable `gLoopedSoundOn` to the opposite state.

The next line calls an application-defined routine which, depending on the value in `gLoopedSoundOn`, toggles the button title between "Switch Looped Sound On" and "Switch Looped Sound Off" and toggles the "Record Sound Resource" and "Record Sound File" buttons between the deactivated and activated states.

Depending on the value in `gLoopedSoundOn`, the if/else block will be sending either the `freqCmd` command or the `quietCmd` command to the channel on which the looped sound is installed. In both of these commands, `param1` should be set to 0.

If the value in `gLoopedSoundOn` is true, the `cmd` field of a sound command structure is assigned `freqCmd` and the `param2` field is assigned a value (60 decimal) which equates to middle C. (The `freqCmd` command changes the frequency (or pitch) of a sound. Also, if no sound is currently playing, `freqCmd` causes the Sound Manager to begin playing at the specified

frequency. If, however, no voice is installed in the channel, no sound is produced. (A voice was installed in the channel to which the command will be sent by the application-defined routine DoLoopedSoundSetUp.)

If the value in gLoopedSoundOn is false, the cmd field of a sound command structure is assigned quietCmd and the param2 field is assigned 0. (The quietCmd command stops the sound that is currently playing, and should be sent using SndDoImmediate.)

SndDoImmediate is called to send the command specified in the second parameter to the sound channel specified in the first parameter. If SndDoImmediate returns an error, an error alert is presented.

DoUnloopedSoundAsync

DoUnloopedSoundAsync uses the ASynchSoundLib function AS_PlayID to play a 'snd ' resource asynchronously.

AS_PlayID is called to play the 'snd ' resource specified in the first parameter. Since no further control over the playback is required, NULL is passed in the second parameter. (Recall that, if you pass a pointer to a variable in the second parameter, AS_PlayID returns a reference number in that parameter. That reference number may be used to gain more control over the playback process. If you simply want to trigger a sound and let it to run to completion, you pass nil in the second parameter, in which case a reference number is not returned by AS_PlayID.)

If AS_PlayID returns the "no channels currently available" error, an error alert is presented advising of that specific condition. If any other error is returned, a more generalised error alert is presented.

When the sound has finished playing, ASynchSoundLib advises the application by setting the application's "attention" flag to true. Recall that this will cause the ASynchSoundLib function AS_CloseChannel to be called to free up the relevant ASstructure, close the relevant sound channel, clear the "attention" flag, and draw some text in the group box to the right of the image well to indicate to the user that AS_CloseChannel has just been called.

Note: The 701-byte 'snd ' resource being used contains one command only (bufferCmd). The compressed sound header indicates MACE 6:1 compression. The loopStart field of the compressed sound header contains 3704 and the loopEnd field contains 3705. The 8-bit mono sound was sampled at 22kHz.

DoSpeakStringAsync

DoSpeakStringAsync is identical to the function doSpeakStringSync except that, in this function, SpeechBusy is not used to delay the function returning until the speech activity spawned by SpeakString has run its course.

DoSetUpDialog

DoSetUpDialog sets up the graphic alignment, graphic offset, and text placement for the bevel buttons. It then deactivates any bevel buttons relating to sound features not available on the machine on which the program is running.

DoAdjustItems

DoAdjustItems toggles the "Switch looped Sound" button between on and off, and the "Record Sound Resource" and "Record Sound File" buttons between activated and deactivated, according to the value in gLoopedSoundOn.