21 Version 1.1

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.

Sound Control Panel. For playback, the user can select a sound output device, and set certain characteristics of the selected device, using the Sound control panel. The 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.

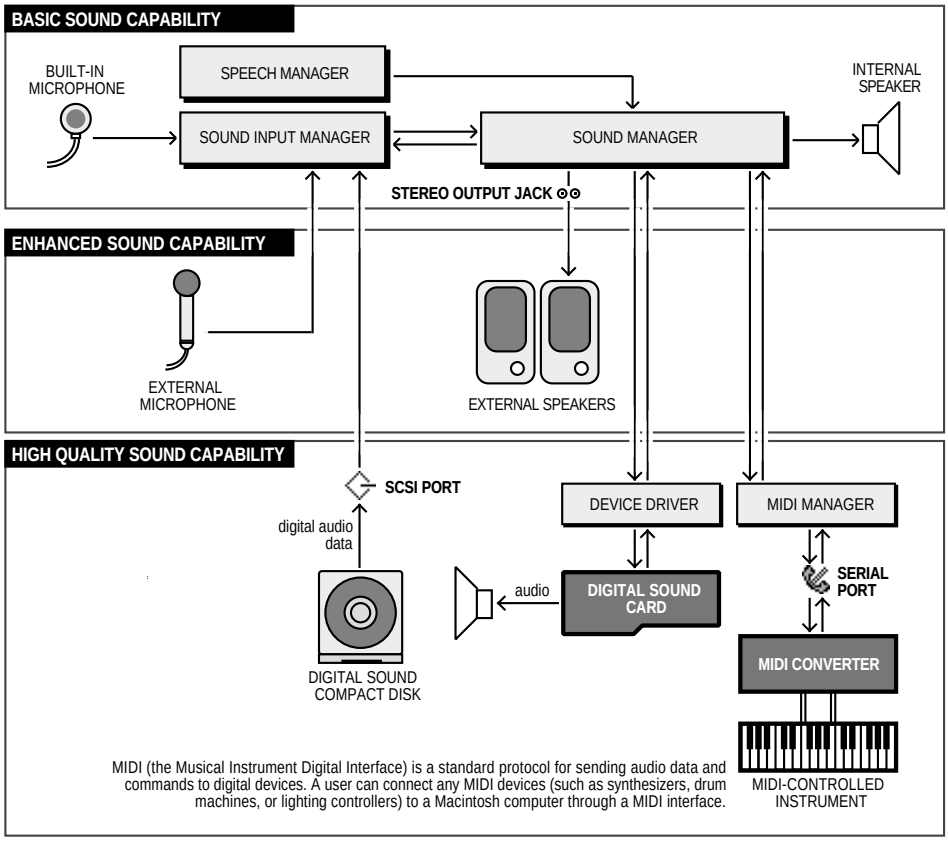

FIG 1 - SOUND CAPABILITIES OF MACINTOSH COMPUTERS

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.

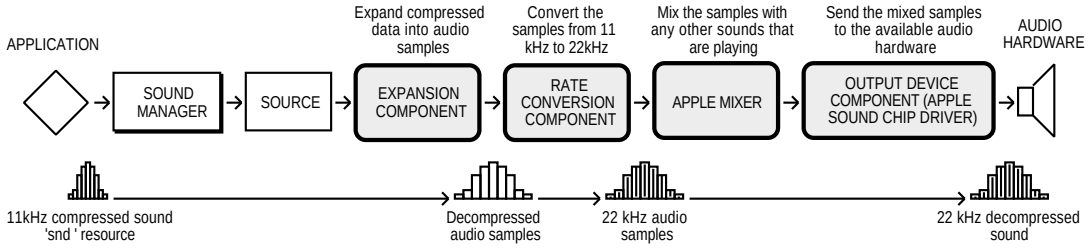

FIG 2 - A TYPICAL SOUND COMPONENT CHAIN

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]

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

---

[1]A term closely associated with the subject of codecs is **MACE** (**Macintosh Audio Compression and Expansion**). MACE is a collection of Sound Manager routines 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.

[2]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.

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** routines 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 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:

```
struct SndChannel
{
     struct SndChannel  *nextChan;          // Pointer to next channel.
     Ptr                firstMod;           // (Used internally.)
     SndCallBackUPP     callBack;           // Pointer to callback procedure.
     long               userInfo;           // Free for application's use.
     long               wait;               // (Used internally.)
     SndCommand         cmdInProgress;      // (Used internally.)
     short              flags;              // (Used internally.)
     short              qLength;            // (Used internally.)
     short              qHead;              // (Used internally.)
     short              qTail;              // (Used internally.)
     SndCommand         queue[stdQLength];  // (Used internally.)
}

typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

Multiple Sound Channels

Except on basic Macintosh models such as the Classic, 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:

```
struct SndCommand
{
  unsigned short  cmd;     // Command number.
  short           param1;  // First parameter.
  long            param2;  // Second parameter.
};

typedef struct SndCommand SndCommand;
```

[3]Do 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 routines to open the Finder sound file and then by using the `SndPlay` function to play the single sound resource contained in it.

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

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

```
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 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 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.

[4]The 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.

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` routine. 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` routine

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.

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` routine. 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` routine.

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 22 — 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, titled AsynchSoundLib, 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 a straightforward and uncomplicated interface for asynchronous sound playback, is documented following the Constants, Data Types, and Routines 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 routines that allow your application to record sounds from the user and store them in memory or in a file. When you call these routines, the Sound Input Manager presents the sound recording dialog box shown at Fig 3.

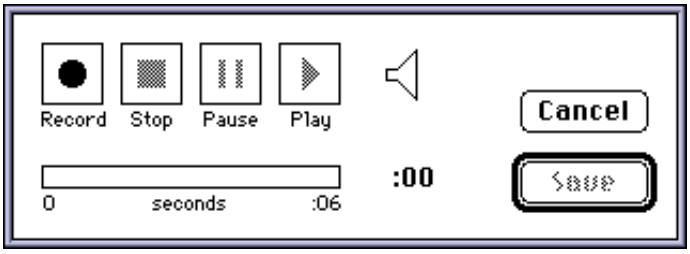

FIG 3 - SOUND RECORDING DIALOG

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 routines 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'`	Lowest quality, least storage space.

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 Routines

Constants

Gestalt Sound Attributes Selector and Response Bits

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

gestaltSpeechAttr           'ttsc'     // Speech Manager attributes.
gestaltSpeechMgrPresent     = 0        // Speech Manager exists.
gestaltSpeechHasPPCGlue     = 1        // Native PPC glue for Speech Manager API exists.
```

Recording Qualities

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

Typical Sound Commands

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

Data Types

Sound Channel Record

```
struct SndChannel
{
  struct SndChannel  *nextChan;          // Pointer to next channel.
  Ptr                firstMod;           // (Used internally.)
  SndCallBackUPP     callBack;           // Pointer to callback procedure.
  long               userInfo;           // Free for application's use.
  long               wait;               // (Used internally.)
  SndCommand         cmdInProgress;      // (Used internally.)
  short              flags;              // (Used internally.)
  short              qLength;            // (Used internally.)
  short              qHead;              // (Used internally.)
  short              qTail;              // (Used internally.)
  SndCommand         queue[stdQLength];  // (Used internally.)
}

typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

Sound Command Record

```
struct SndCommand
{
   unsigned short     cmd;              // Command number.
   short              param1;           // First parameter.
   long               param2;           // Second parameter.
};

typedef struct SndCommand SndCommand;
```

Routines

Playing Sound Resources

```
void    SysBeep(short duration);
OSErr   SndPlay(SndChannelPtr chan,SndListHandle sndHdl,Boolean async);
```

Playing From Disk

```
OSErr   SndStartFilePlay(SndChannelPtr chan,short fRefNum,short resNum,long bufferSize,
        void *theBuffer,AudioSelectionPtrtheSelection,FilePlayCompletionUPP theCompletion,
        Boolean async);
OSErr   SndPauseFilePlay(SndChannelPtr chan);
OSErr   SndStopFilePlay(SndChannelPtr chan,Boolean quietNow);
```

Allocating and Releasing Sound Channels

```
OSErr   SndNewChannel(SndChannelPtr *chan,short synth,long init,SndCallBackUPP userRoutine);
OSErr   SndDisposeChannel(SndChannelPtr chan,Boolean quietNow);
```

Sending Commands to a Sound Channel

```
OSErr   SndDoCommand(SndChannelPtr chan,const SndCommand *cmd,Boolean noWait);
OSErr   SndDoImmediate(SndChannelPtr chan,const SndCommand *cmd);
```

Recording Sounds

```
OSErr   SndRecord(ModalFilterUPP filterProc,Point corner,OSType quality,
        SndListHandle *sndHandle);
OSErr   SndRecordToFile(ModalFilterUPP filterProc,Point corner,OSType quality,short fRefNum);
```

Generating Speech

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

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 routines 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 **ASLRecord**, with each channel. Each ASLRecord includes the following fields:

```
SndChannel  channel;      // The sound channel.
SInt32      refNum;       // Reference number.
Handle      sound;        // The sound.
char        handleState;  // State to which to restore the sound handle.
Boolean     inUse;        // Is this ASLRecord currently in use?
```

```
OSErr  ASLinitialise   (attnFlag,numChannels);

Boolean    *attnFlag;     Pointer to application's "attention" flag global variable.
SInt16     numChannels;   Number of channels required to be open simultaneously.  If 0 is
                          specified, numChannels 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 ASLRecords equal to the requested number of sound channels.

```
OSErr ASLplayID  (resID,refNum);

SInt16     resID          Resource ID of the 'snd ' resource.
SInt32     *refNum        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, `ASLplayID` and its sister routine `ASLplayHandle` (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, `ASLplayID` attempts to load the specified `'snd '` resource. If successful, the handle state is saved for later restoration, and the handle is made unpurgeable. The function then gets a reference number and a pointer to the next free ASLRecord. A sound channel is then allocated via a call to `SndNewChannel` and the associated ASLRecord 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, ASLPlayID returns the reference number associated with the channel (if the caller wants it).

```
OSErr  ASLplayHandle  (sound,refNum);

Handle     sound         A handle to the sound to be played.
SInt32     *refNum       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 ASLplayHandle routine is similar to ASLplayID, except that it supports a special case: You can pass ASLplayHandle a NULL handle. This causes ASLplayHandle 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 ASLgetChannel, below.)

If a handle is provided, its current state is saved for later restoration before it is made unpurgeable. ASLplayHandle then gets a reference number and a pointer to a free ASLRecord. A sound channel is then allocated via a call to SndNewChannel and the associated ASLRecord 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).

```
OSErr  ASLgetChannel  (refNum,channel);

Sint32         refNum       Reference number.
SndChannelPtr  *channel     A pointer to a SoundChannelPtr.

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

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

ASLgetChannel 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 ASLplayHandle, specifying NULL in the first parameter. (This causes SHPlayByHandle to open a sound channel but not call SndPlay.) Armed with the returned reference number associated with that channel, you then call ASLgetChannel 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.

```
void  ASLcloseChannel  (void);
```

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 ASLRecords looking for records 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 records for later use and closes the associated sound channel.

```
void  ASLcloseDown  (void);
```

ASLcloseDown checks that AsynchSoundLib was previously initialised, stops all current playback, calls ASLcloseChannel to close open sound channels, and disposes of the associated ASLRecords.

Demonstration Program

```
// ############################################################################################
// Sound.c
// ############################################################################################
//
// This program opens a modal dialog containing eight button controls arranged in two
// groups, namely, a synchronous sound group and an asynchronous sound group.  Clicking
// on the 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.
//
// The program utilises the following resources:
//
// •  A 'DLOG' resource and associated 'DITL' and 'dctb' resources (all purgeable).
//
// •  Three 'snd ' resources, one for synchronous playback (purgeable), one for looped
//    asynchronous playback (unpurgeable), and one for unlooped asynchronous playback
//    (purgeable).
//
// •  Two 'cicn' resources (purgeable) used to provide an animated display which halts
//    during synchronous playback and continues during asynchronous playback.
//
// •  Three 'STR#' resources containing error message strings and "speak text" strings
//    (all purgeable).
//
// •  Two 'ALRT' resources (purgeable) for displaying error messages.
//
// In addition, the function doPlayFile utilises the file "soundfile.aiff".
//
// Each time is is invoked, the function doRecordResource creates a new 'snd' resource
// with a unique ID in the application's resource fork.
//
// When first invoked, the function doRecordFile creates the file "test.aiff" in the
// chap21cw_demo folder.  All subsequent record-to-file is to this file.
//
// ############################################################################################

// ……… includes

#include <SoundInput.h>
#include <Speech.h>
#include <Gestalt.h>
#include <Fonts.h>
#include <Resources.h>
#include <ToolUtils.h>
#include <SegLoad.h>
#include <Files.h>

// ……… defines

```

```
#define rDialog                 128
#define  iQuit                  1
#define  iPlayResource          2
#define  iPlayFile              3
#define  iRecordResource        4
#define  iRecordFile            5
#define  iSpeakTextSync         6
#define  iLoopedSound           7
#define  iUnloopedSound         8
#define  iSpeakTextAsync        9
#define  iSynchSoundRect        10
#define  iAsynchSoundRect       11
#define rPlaySoundResource      8192
#define rLoopedSound            8193
#define rUnloopedSound          8194
#define rSpeechStrings          130
#define rErrorAlert             129
#define rErrorStrings           128
#define  eOpenDialogFail        1
#define  eLoopedSoundSetUp      2
#define  eCannotInitialise      3
#define  eGetResource           4
#define  eNoChannelsAvailable   5
#define  ePlaySound             6
#define  eMemory                7
#define rErrorAlertWithCode     130
#define rErrorStringsWithCode   129
#define  eSndPlay               1
#define  ePlayFile              2
#define  eSndRecord             3
#define  eWriteResource         4
#define  eRecordFile            5
#define  eSpeakString           6
#define  eSndDoImmediate        7
#define rColourIcon1            128
#define rColourIcon2            129
#define kMaxChannels            8
#define kOutOfChannels          1

// ……… global variables

Boolean       gDone;
DialogPtr     gDialogPtr;
SInt16        gAppResFileRefNum;
Boolean       gColorQuickDrawPresent = false;
Boolean       gHasSoundPlayDoubBuff;
Boolean       gHasSoundInputDevice;
Boolean       gHasSpeechmanager;
Boolean       gHasMultiChannel;
Boolean       gLoopedSoundOn = false;
SInt32        gLoopedSoundRefNum;
SndChannelPtr gLoopedSoundChannel;
CIconHandle   gColourIconHdl1;
CIconHandle   gColourIconHdl2;

// ……… AsyncSoundLib attention flag

Boolean       gCallASLcloseChannel = false;

// ……… function prototypes

void        main                (void);
void        doInitManagers      (void);
void        doCheckSoundEnv     (void);
void        doInitialiseASL     (void);
Boolean     doLoopedSoundSetUp  (void);
void        eventLoop           (void);
void        doDialogHit         (SInt16);
void        doPlayResource      (void);
void        doPlayFile          (void);
void        doRecordResource    (void);
void        doRecordFile        (void);
void        doSpeakStringSync   (void);
void        doLoopedSoundAsync  (void);
void        doUnloopedSoundAsync (void);
void        doSpeakStringAsync  (void);
void        doSetUpDialog       (void);
```

```
pascal void drawDialog            (DialogPtr,SInt16);
void        doAdjustItems         (void);
void        doErrorAlert          (SInt16);
void        doErrorAlertWithCode  (SInt16,SInt16);

// ……… AsyncSoundLib function prototypes

OSErr       ASLinitialise         (Boolean *,SInt16);
OSErr       ASLgetChannel         (SInt32,SndChannelPtr *);
OSErr       ASLplayID             (SInt16, SInt32 *);
OSErr       ASLplayHandle         (Handle,SInt32 *);
void        ASLcloseChannel       (void);
void        ASLcloseDown          (void);

// ########################################################################### main

void main(void)
{
  OSErr   osErr;
  SInt32  response;

  // ……… check for Color QuickDraw

  osErr = Gestalt(gestaltQuickdrawVersion,&response);
  if(response >= gestalt8BitQD)
    gColorQuickDrawPresent = true;

  // ……… initialise managers

  doInitManagers();

  // ……… save reference number of application's resource file

  gAppResFileRefNum = CurResFile();

  // ……… check for sound recording equipment and speech capabilities

  doCheckSoundEnv();

  // ……… open and set up modal dialog, get colour icons

  if(!(gDialogPtr = GetNewDialog(rDialog,NULL,(WindowPtr) -1)))
  {
    doErrorAlert(eOpenDialogFail);
    ExitToShell();
  }

  SetPort(gDialogPtr);

  doSetUpDialog();

  if(gColorQuickDrawPresent)
  {
    gColourIconHdl1 = GetCIcon(rColourIcon1);
    gColourIconHdl2 = GetCIcon(rColourIcon2);
  }

  // ……… initialize AsychSoundLib

  doInitialiseASL();

  // ……… set up looped sound

  if(gHasMultiChannel)
  {
    if(!doLoopedSoundSetUp())
    {
      doErrorAlert(eLoopedSoundSetUp);
      ASLcloseDown();
      ExitToShell();
    }
  }

  // ……… enter event loop

  eventLoop();
}
```

```
229
230 // ############################################################################ doInitManagers
231
232 void  doInitManagers(void)
233 {
234   MaxApplZone();
235   MoreMasters();
236
237   InitGraf(&qd.thePort);
238   InitFonts();
239   InitWindows();
240   InitMenus();
241   TEInit();
242   InitDialogs(NULL);
243
244   InitCursor();
245   FlushEvents(everyEvent,0);
246 }
247
248 // ############################################################################ doCheckSoundEnv
249
250 void  doCheckSoundEnv(void)
251 {
252   OSErr   osErr;
253   SInt32  response;
254
255   osErr = Gestalt(gestaltSoundAttr,&response);
256
257   if(osErr == noErr)
258     gHasSoundPlayDoubBuff = BitTst(&response,31 - gestaltSndPlayDoubleBuffer);
259   else
260     gHasSoundPlayDoubBuff = false;
261
262   if(osErr == noErr)
263     gHasSoundInputDevice = BitTst(&response,31 - gestaltHasSoundInputDevice);
264   else
265     gHasSoundInputDevice = false;
266
267   if(osErr == noErr)
268     gHasSpeechmanager = BitTst(&response,31 - gestaltSpeechMgrPresent);
269   else
270     gHasSpeechmanager = false;
271
272   if(osErr == noErr)
273     gHasMultiChannel = BitTst(&response,31 - gestaltMultiChannels);
274   else
275     gHasMultiChannel = false;
276 }
277
278 // ############################################################################ doInitialiseASL
279
280 void  doInitialiseASL(void)
281 {
282   if(ASLinitialise(&gCallASLcloseChannel,kMaxChannels) != noErr)
283   {
284     doErrorAlert(eCannotInitialise);
285     ExitToShell();
286   }
287 }
288
289 // ############################################################################ doLoopedSoundSetUp
290
291 Boolean  doLoopedSoundSetUp(void)
292 {
293   SInt16  error;
294   OSErr   osErr;
295   Handle  soundHdl;
296
297   error = ASLplayHandle(NULL,&gLoopedSoundRefNum);
298   if(error)
299     return(false);
300   else
301   {
302     error = ASLgetChannel(gLoopedSoundRefNum,&gLoopedSoundChannel);
303     if(error)
304       return(false);
305
```

```
    soundHdl = GetResource('snd ',rLoopedSound);
    if(soundHdl)
    {
      HLockHi(soundHdl);
      osErr = SndPlay(gLoopedSoundChannel,(SndListHandle) soundHdl,true);
      if(osErr != noErr)
        return(false);
    }
    else
      return(false);
  }

  return(true);
}

// ############################################################################## eventLoop

void  eventLoop(void)
{
  Rect        theRect, eraseRect;
  Boolean     gotEvent;
  EventRecord eventRec;
  DialogPtr   theDialogPtr;
  SInt16      itemHit;
  SInt32      finalTicks;

  gDone = false;

  SetRect(&theRect,10,273,35,299);
  SetRect(&eraseRect,45,273,125,299);

  while(!gDone)
  {
    if(gCallASLcloseChannel)
    {
      ASLcloseChannel();

      TextFont(geneva);
      TextSize(9);
      MoveTo(45,285);
      DrawString("\pASLcloseChannel");
      MoveTo(45,295);
      DrawString("\pcalled");
    }

    gotEvent = WaitNextEvent(everyEvent,&eventRec,10,NULL);

    if(gotEvent)
    {
      if(IsDialogEvent(&eventRec))
        if(DialogSelect(&eventRec,&theDialogPtr,&itemHit))
          doDialogHit(itemHit);
    }
    else
    {
      if(gColorQuickDrawPresent)
      {
        PlotCIcon(&theRect,gColourIconHdl1);
        Delay(15,&finalTicks);
        PlotCIcon(&theRect,gColourIconHdl2);
        Delay(15,&finalTicks);
        EraseRect(&eraseRect);
      }
    }
  }

  DisposeDialog(gDialogPtr);

  ASLcloseDown();
}

// ############################################################################## doDialogHit

void  doDialogHit(SInt16 item)
{
  switch(item)
  {
```

```
      case iQuit:
        gDone = true;
        break;

      case iPlayResource:
        doPlayResource();
        break;

      case iPlayFile:
        doPlayFile();
        break;

      case iRecordResource:
        doRecordResource();
        break;

      case iRecordFile:
        doRecordFile();
        break;

      case iSpeakTextSync:
        doSpeakStringSync();
        break;

      case iLoopedSound:
        doLoopedSoundAsync();
        break;

      case iUnloopedSound:
        doUnloopedSoundAsync();
        break;

      case iSpeakTextAsync:
        doSpeakStringAsync();
        break;
    }
  }

  // ############################################################################## doPlayResource

  void  doPlayResource(void)
  {
    SndListHandle sndListHdl;
    SInt16        resErr;
    OSErr         osErr;

    sndListHdl = (SndListHandle) GetResource('snd ',rPlaySoundResource);
    resErr = ResError();
    if(resErr != noErr)
      doErrorAlert(eGetResource);

    if(sndListHdl != NULL)
    {
      HLock((Handle) sndListHdl);
      osErr = SndPlay(NULL,sndListHdl,false);
      if(osErr != noErr)
        doErrorAlertWithCode(eSndPlay,osErr);
      HUnlock((Handle) sndListHdl);
      ReleaseResource((Handle) sndListHdl);
    }
  }

  // ################################################################################ doPlayFile

  void  doPlayFile(void)
  {
    OSErr   osErr;
    FSSpec  fileSysSpec;
    SInt16  fileRefNum;

    osErr = FSMakeFSSpec(0,0,"\p:soundfile.aiff",&fileSysSpec);
    if(osErr == noErr)
      osErr = FSpOpenDF(&fileSysSpec,fsRdPerm,&fileRefNum);
    if(osErr == noErr)
      SetFPos(fileRefNum,fsFromStart,0);
    if(osErr == noErr)
      osErr = SndStartFilePlay(NULL,fileRefNum,0,20480,NULL,NULL,NULL,false);
```

```
  if(osErr != noErr)
    doErrorAlertWithCode(ePlayFile,osErr);

  FSClose(fileRefNum);
}

// ########################################################################## doRecordResource

void  doRecordResource(void)
{
  SInt16  oldResFileRefNum;
  Point   topLeft;
  Handle  soundHdl;
  OSErr   osErr, memErr;
  SInt16  theResourceID, resErr;

  oldResFileRefNum = CurResFile();
  UseResFile(gAppResFileRefNum);

  topLeft.v = 40;
  topLeft.h = 250;

  soundHdl = NewHandle(25000);
  memErr = MemError();
  if(memErr != noErr)
  {
    doErrorAlert(eMemory);
    return;
  }

  osErr = SndRecord(NULL,topLeft,siBetterQuality,&(SndListHandle) soundHdl);
  if(osErr != noErr && osErr != userCanceledErr)
    doErrorAlertWithCode(eSndRecord,osErr);
  else
  {
    do
    {
      theResourceID = UniqueID('snd ');
    } while(theResourceID <= 8191 && theResourceID >= 0);

    AddResource((Handle) soundHdl,'snd ',theResourceID,"\pTest");
    resErr = ResError();
    if(resErr == noErr)
      UpdateResFile(gAppResFileRefNum);
    resErr = ResError();
    if(resErr != noErr)
      doErrorAlertWithCode(eWriteResource,resErr);
  }

  UseResFile(oldResFileRefNum);
}

// ############################################################################ doRecordFile

void  doRecordFile(void)
{
  Point   topLeft;
  OSErr   osErr;
  FSSpec  fileSysSpec;
  SInt16  fileRefNum;

  topLeft.v = 40;
  topLeft.h = 250;

  osErr = FSMakeFSSpec(0,0,"\p:test.aiff",&fileSysSpec);
  if(osErr == fnfErr)
    osErr = FSpCreate(&fileSysSpec,'????','AIFF',smSystemScript);
  if(osErr == noErr)
    osErr = FSpOpenDF(&fileSysSpec,fsWrPerm,&fileRefNum);
  if(osErr == noErr)
    SetFPos(fileRefNum,fsFromStart,0);
  if(osErr == noErr)
    osErr = SndRecordToFile(NULL,topLeft,siBetterQuality,fileRefNum);
  if(osErr != noErr && osErr != userCanceledErr)
    doErrorAlertWithCode(eRecordFile,osErr);

  FSClose(fileRefNum);
```

```
}

// ########################################################################## doSpeakStringSync

void  doSpeakStringSync(void)
{
  SInt16  activeChannels;
  Str255  theString;
  OSErr   resErr, osErr;

  activeChannels = SpeechBusy();

  GetIndString(theString,rSpeechStrings,1);
  resErr = ResError();
  if(resErr != noErr)
  {
    doErrorAlert(eGetResource);
    return;
  }

  osErr = SpeakString(theString);
  if(osErr != noErr)
    doErrorAlertWithCode(eSpeakString,osErr);

  while(SpeechBusy() != activeChannels)
    ;
}

// ########################################################################## doLoopedSoundAsync

void  doLoopedSoundAsync(void)
{
  SndCommand  sndCommand;
  OSErr       osErr;

  gLoopedSoundOn = !gLoopedSoundOn;

  doAdjustItems();

  sndCommand.param1 = 0;

  if(gLoopedSoundOn)
  {
    sndCommand.cmd    = freqCmd;
    sndCommand.param2 = 0x3C;
  }
  else
  {
    sndCommand.cmd    = quietCmd;
    sndCommand.param2 = 0;
  }

  osErr = SndDoImmediate(gLoopedSoundChannel,&sndCommand);
  if(osErr != noErr)
    doErrorAlertWithCode(eSndDoImmediate,osErr);
}

// ########################################################################## doUnloopedSoundAsync

void  doUnloopedSoundAsync(void)
{
  SInt16  error;

  error = ASLplayID(rUnloopedSound,NULL);
  if(error == kOutOfChannels)
    doErrorAlert(eNoChannelsAvailable);
  else
    if(error != noErr)
      doErrorAlert(ePlaySound);
}

// ########################################################################## doSpeakStringAsync

void  doSpeakStringAsync(void)
{
  Str255  theString;
  OSErr   resErr, osErr;
```

```
614
615   GetIndString(theString,rSpeechStrings,2);
616   resErr = ResError();
617   if(resErr != noErr)
618   {
619     doErrorAlert(eGetResource);
620     return;
621   }
622
623   osErr = SpeakString(theString);
624   if(osErr != noErr)
625     doErrorAlertWithCode(eSpeakString,osErr);
626 }
627
628 // ############################################################################ doSetUpDialog
629
630 void  doSetUpDialog(void)
631 {
632   SInt16  itemType;
633   Handle  itemHdl;
634   Rect    itemRect;
635
636   GetDialogItem(gDialogPtr,iSynchSoundRect,&itemType,&itemHdl,&itemRect);
637   SetDialogItem(gDialogPtr,iSynchSoundRect,itemType,(Handle) drawDialog,&itemRect);
638
639   if(!gHasSoundPlayDoubBuff)
640   {
641     GetDialogItem(gDialogPtr,iPlayFile,&itemType,&itemHdl,&itemRect);
642     HiliteControl((ControlHandle) itemHdl,255);
643   }
644
645   if(!gHasSoundInputDevice)
646   {
647     GetDialogItem(gDialogPtr,iRecordResource,&itemType,&itemHdl,&itemRect);
648     HiliteControl((ControlHandle) itemHdl,255);
649     GetDialogItem(gDialogPtr,iRecordFile,&itemType,&itemHdl,&itemRect);
650     HiliteControl((ControlHandle) itemHdl,255);
651   }
652
653   if(!gHasSpeechmanager)
654   {
655     GetDialogItem(gDialogPtr,iSpeakTextSync,&itemType,&itemHdl,&itemRect);
656     HiliteControl((ControlHandle) itemHdl,255);
657     GetDialogItem(gDialogPtr,iSpeakTextAsync,&itemType,&itemHdl,&itemRect);
658     HiliteControl((ControlHandle) itemHdl,255);
659   }
660
661   if(!gHasMultiChannel)
662   {
663     GetDialogItem(gDialogPtr,iLoopedSound,&itemType,&itemHdl,&itemRect);
664     HiliteControl((ControlHandle) itemHdl,255);
665   }
666 }
667
668 // ############################################################################ drawDialog
669
670 pascal void  drawDialog(DialogPtr dialogPtr,SInt16 theItem)
671 {
672   SInt16    itemType;
673   Handle    itemHdl;
674   Rect      itemRect;
675   SInt16    buttonOval;
676
677   GetDialogItem(dialogPtr,iSynchSoundRect,&itemType,&itemHdl,&itemRect);
678   FrameRect(&itemRect);
679   GetDialogItem(dialogPtr,iAsynchSoundRect,&itemType,&itemHdl,&itemRect);
680   FrameRect(&itemRect);
681   GetDialogItem(dialogPtr,iQuit,&itemType,&itemHdl,&itemRect);
682   InsetRect(&itemRect,-4,-4);
683   PenSize(3,3);
684   buttonOval = (itemRect.bottom - itemRect.top) / 2 + 2;
685   FrameRoundRect(&itemRect,buttonOval,buttonOval);
686 }
687
688 // ############################################################################ doAdjustItems
689
690 void  doAdjustItems(void)
```

```
691  {
692    SInt16  itemType, a;
693    Handle  itemHdl;
694    Rect    itemRect;
695
696    GetDialogItem(gDialogPtr,iLoopedSound,&itemType,&itemHdl,&itemRect);
697    if(gLoopedSoundOn)
698      SetControlTitle((ControlHandle) itemHdl,"\pSwitch Looped Sound Off");
699    else
700      SetControlTitle((ControlHandle) itemHdl,"\pSwitch Looped Sound On");
701
702    for(a=iRecordResource;a<iRecordFile+1;a++)
703    {
704      GetDialogItem(gDialogPtr,a,&itemType,&itemHdl,&itemRect);
705      if(gLoopedSoundOn)
706        HiliteControl((ControlHandle) itemHdl,255);
707      else
708        HiliteControl((ControlHandle) itemHdl,0);
709    }
710  }
711
712  // ############################################################################## doErrorAlert
713
714  void doErrorAlert(SInt16 stringIndex)
715  {
716    Str255  errorString;
717
718    GetIndString(errorString,rErrorStrings,stringIndex);
719    ParamText(errorString,NULL,NULL,NULL);
720    StopAlert(rErrorAlert,NULL);
721  }
722
723  // ######################################################################## doErrorAlertWithCode
724
725  void doErrorAlertWithCode(SInt16 stringIndex,SInt16 resultCode)
726  {
727    Str255  errorString, resultCodeString;
728
729    GetIndString(errorString,rErrorStringsWithCode,stringIndex);
730    NumToString((SInt32) resultCode,resultCodeString);
731
732    ParamText(errorString,resultCodeString,NULL,NULL);
733    StopAlert(rErrorAlertWithCode,NULL);
734  }
735
736  // ##############################################################################################
```

Demonstration Program Comments

When this program is run, the user should click on the various buttons to record and play back sound resources and sound files and to play back the provided "speak text" strings. On machines with Color QuickDraw, the user should observe the effects of asynchronous and synchronous playback on the "working man" icon at the lower left of the dialog. The user should also observe that the text "ASLcloseChannel called" appears briefly at the bottom of the dialog when AsynchSoundLib sets the application's "attention" flag to true, thus causing the application to call the AsynchSoundLib function ASLcloseChannel.

Note that the doRecordResource function saves recorded sounds as 'snd ' resources with unique IDs in the resource fork of the application (Sound). 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.

#define

Lines 75-86 establish constants relating to the dialog's resource ID and items. Lines 87-90 establish constants relating to sound resource IDs and the ID of the 'STR#' resource containing the "speak text" strings. Lines 91-108 establish constants relating to error alert 'ALRT' resource IDs and associated error strings. Lines 109-110 establish constants relating to colour icon 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 routine ASLplayID returns a "no channels available" error.

Global Variables

gDone controls program termination. gDialogPtr will be assigned the address of the dialog's dialog record. The application's resource file reference number will be saved to gAppResFileRefNum at startup. gColorQuickDrawPresent will be set to true if Color QuickDraw is present.

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 button. gLoopedSoundRefNum will be assigned the reference number returned by a call to the AsynchSoundLib routine ASLplayHandle. gLoopedSoundChannel will be assigned the pointer to the sound channel record returned by a call to the AsynchSoundLib routine ASLgetChannel.

gColourIconHdl1 and gColourIconHdl2 will be assigned handles to the two colour icon resources.

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

main

The main function checks for Color QuickDraw (Lines 175-177), initialises the system software managers (Line 181), saves the reference number of the application's resource file (Line 185), checks the sound environment and sets the associated global variables accordingly (Line 189), opens and sets up the dialog (Lines 193-201), and gets two colour icons if Color QuickDraw is present (Lines 203-207).

AsynchSoundLib is then initialised (Line 211). If multi-channel playback is available (Line 215), an application-defined function is called to set up the looped sound playback (Line 217). If this call is not successful, an error alert is displayed, the AsynchSoundLib routine ASLcloseDown is called and the program terminates (Lines 219-221).

> Note: Line 215 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.

At Line 227, the main event loop is entered.

doCheckSoundEnv

doCheckSoundEnv checks for play-from-disk capability (Line 259), recording capability (Line 263), speech capability (Line 268), and multi-channel playback capability (Line 273), and sets the associated global variables accordingly.

> Note: doCheckSoundEnv uses the function BitTst to determine whether the appropriate bit in Gestalt's response is set to 1. Bit numbering with BitTst is the opposite of the usual MC680x0 numbering scheme used by Gestalt. Thus the bit to be tested must be subtracted from 31. This is illustrated in the following:
>
> ```
> Bit numbering as used in BitTst
> 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
> Bit as numbered in MC69000 CPU operations, and used by Gestalt
> 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0  15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
>
> gestaltHasSoundInputDevice = 5
> 31 - 5 = 26
> ```

doInitialiseASL

doInitialiseASL initialises the AsynchSoundLib. More specifically, it calls the AsynchSoundLib routine ASLinitialise (Line 282) and passes to AsynchSoundLib the address of

the application's "attention" flag (gASLcloseChannel), together with the requested number of channels.

If ASLinitialise returns a non-zero value, an error alert is displayed and the program terminates (Lines 284-285).

doLoopedSoundSetUp

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

First, at Line 297, the AsynchSoundLib routine ASLplayHandle is called with NULL passed as the first parameter. (This causes ASLplayHandle 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 ASLplayHandle.

If the call to ASLplayHandle is successful, Line 302 calls the AsynchSoundLib routine ASLgetChannel, passing the reference number returned by ASLplayHandle in the first parameter and receiving a pointer to the sound channel in the second parameter.

If the call to ASLgetChannel is successful, Line 306 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 (Line 309). SndPlay is then called with true passed as 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.

> Note: 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

eventLoop contains the main event loop.

Line 332 sets the global variable gDone to false. When this variable is set to true, the program will terminate. Lines 334-335 define two rectangles which will be used in the drawing of the colour icons and in erasing some text at the bottom of the dialog.

The event loop is entered at Line 337.

Within the loop, the "attention" flag required by AsynchSoundLib is checked. If AsynchSoundLib has set it to true (Line 339), the AsynchSoundLib function ASLcloseChannel is called (Line 341) to free up the relevant ASLRecord, close the relevant sound channel, and clear the "attention" flag. In addition, some text is drawn at the bottom of the dialog to indicate to the user that ASLcloseChannel has just been called (Lines 343-348).

If WaitNextEvent retrieves an event other than a NULL event (Line 353), IsDialogEvent is called to determine whether the event was within a dialog. If so, DialogSelect is called to determine whether one of the dialog's buttons was clicked (Line 356). If so, the application-defined function doDialogHit is called to further process the item hit event.

If a null event was returned by WaitNextEvent (Line 361), and if Color QuickDraw is present (Line 361), Lines 363-367 use the two colour icons to draw the two frames of "working man" animation and erase the area in which the "ASLcloseChannel called" may have been drawn.

When gDone is set to true, the event loop exits, the dialog is disposed of (Line 372), and the AsynchSoundLib function ASLcloseDown is called to stop all current playback, close open sound channels, and dispose of the associated ASLRecords (Line 374).

doDialogHit

doDialogHit switches according to the received item number and calls the appropriate application-defined function 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.

Line 429 attempts to load the resource. If the subsequent call to ResError indicates an error, an error alert is presented (Lines 430-432).

If the load was successful (Line 434), the sound handle is locked prior to a call to SndPlay (Lines 436-437). 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 as 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 (Lines 438-439).

When SndPlay returns, Lines 440-441 unlock the sound handle and release the resource.

doPlayFile

doPlayFile uses SndStartFilePlay to play a specified sound file.

Line 453 converts the directory specification shown into an FSSpec record. The pointer to the FSSpec record returned by FSMakeFSSpec is passed in the first parameter of a call to FSpOpenDF at Line 455. FSpOpenDF opens the file's data fork and receives the file reference number in its third parameter. SetFPos (Line 457) positions the file mark to the beginning of the file.

The file reference number is passed as the second parameter in the call to SndStartFilePlay at Line 459. The parameters passed to SndStartFilePlay are as follows:

- NULL 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.
- NULL in the theBuffer parameter causes the Sound Manager to internally allocate two relocatable blocks, each of which is half the size of bufferSize.
- NULL in the theSelection parameter means the entire sound will be played.
- NULL in the theCompletion parameter means that there is no completion routine 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, Line 459 presents an error alert.

Line 463 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.

Lines 476-477 save the current resource file reference number and set 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).)

Lines 479-480 establish the location for the top left corner of the sound recording dialog.

Line 482 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 (Lines 483-488);

SndRecord (Line 490) 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 at Line 482, 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, Lines 495-498 first find an acceptable unique resource ID for the resource. (For the System 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 recources to the System file.). The call to AddResource at Line 500 causes the Resource Manager to regard the relocatable block containing the sound as a 'snd ' resource. If the call is successful, Line 503 writes the changed resource map and the 'snd ' resource to disk. If an error occurs, an error alert is presented (Lines 505-506)

Line 509 restores the resource file saved at Line 476 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.

Lines 521-522 establish the location for the top left corner of the sound recording dialog.

At Line 524, FSMakeFSSpec converts the directory specification passed in its third parameter into an FSSpec record. If FSMakeFSSpec returns fnfErr (file not found), Line 526 creates a new file of type 'AIFF'. Line 528 opens the file's data fork and Line 530 positions the file mark to the beginning of the file.

SndRecordToFile (Line 532) 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 (Lines 533-534).

Line 536 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 psuedo-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, Line 547 saves the number of speech channels that are active immediately before the call to SpeakString.

Line 549 loads the first string from the specified 'STR#' resource. If an error occurs, a dialog is presented and the function returns (Lines 550-555).

At Line 557, SpeakString, which automatically allocates a speech channel, is called to speak the string. If SpeakString returns an error, an error alert is presented (Lines 558-559).

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 that the number saved at Line 547. Accordingly, the while loop entered at Line 561 continues until the number of open speech channels is equal to the number saved at Line 547. 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 function doLoopedSoundSetUp, and on which doLoopedSoundSetUp has already installed a voice.

Line 572 toggles the Boolean global variable gLoopedSoundOn to the opposite state.

Line 574 calls an application-defined function which, depending on the 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 disabled and enabled states.

Depending on the value in gLoopedSoundOn, Lines 578-589 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 (Line 576).

If the value in gLoopedSoundOn is true (Line 478), the cmd field of a sound command record is assigned freqCmd and the param2 field is assigned a value (60 decimal) which equates to middle C (Lines 580-581). (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 function doLoopedSoundSetUp.)

If the value in gLoopedSoundOn is false (Line 583), the cmd field of a sound command record 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 using SndDoImmediate.)

Line 589 calls SndDoImmediate 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 (Lines 590-591).

doUnloopedSoundAsync

doUnloopedSoundAsync uses the AsynchSoundLib routine ASLplayID to play a 'snd ' resource asynchronously.

At Line 600, ASLplayID 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, ASLplayID 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 NULL in the second parameter, in which case a reference number is not returned by ASLplayID.)

If ASLplayID returns the "no channels currently available" error, an error alert is presented advising of that specific condition (Lines 601-602). If any other error is returned, a more generalised error alert is presented (Lines 603-605).

When the sound has finished playing, ASynchSoundLib advises the application by setting the application's "attention" flag to true. Recall from Lines 339-341 that this will cause the AsynchSoundLib function ASLcloseChannel to be called to free up the relevant ASLRecord, close the relevant sound channel, clear the "attention" flag, and draw some text at the bottom of the dialog to indicate to the user that ASLcloseChannel has just been called (Lines 343-348).

> 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 and drawDialog

doSetUpDialog first installs an application-defined draw function (drawDialog) in one of the dialog's user items. It then disables any buttons relating to sound features not available on the machine on which the program is running. drawDialog is called whenever the dialog gets an update event. It draws the two group rectangles and the bold outline around the "Done" button.

doAdjustItems

doAdjustItems toggles the "Switch looped Sound" button between on and off, and the "Record Sound Resource" and "Record Sound File" buttons between enabled and disabled, according to the value in gLoopedSoundOn.

doErrorAlert and doErrorAlertWithCode

doErrorAlert and doErrorAlertWithCode retrieve the strings associated with the various error conditions and present an alert displaying the string. doErrorAlertWithCode also displays the error code number itself.