

24

BASIC SOUND AND SPEECH

Demonstration Program: SoundAndSpeech

Introduction to Sound

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

Audio Hardware and Sound-Related System Software

The **audio hardware** includes a speaker or speakers, a microphone, and one or more integrated circuits that convert digital data to analog signals and vice versa.

The sound-related system software managers are as follows:

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

Sound Input and Output Capabilities

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

- The playing back of digitally recorded sounds. (Digitally recorded sound is referred to as **sampled** sound.
- The playing back of simple sequences of notes or of complex waveforms.
- The recording of sampled sounds.
- The conversion of text to spoken words.
- The mixing and synchronisation of multiple channels of sampled sounds.
- The compression and decompression of sound data.

- The integration and synchronisation of sound production with the display of video and still images. (For example, the Sound Manager is used by QuickTime to handle all the sound data in a QuickTime movie.)

Basic and Enhanced Sound Capabilities

Users can enhance sound playback and recording quality by substituting better speakers and microphones. 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.

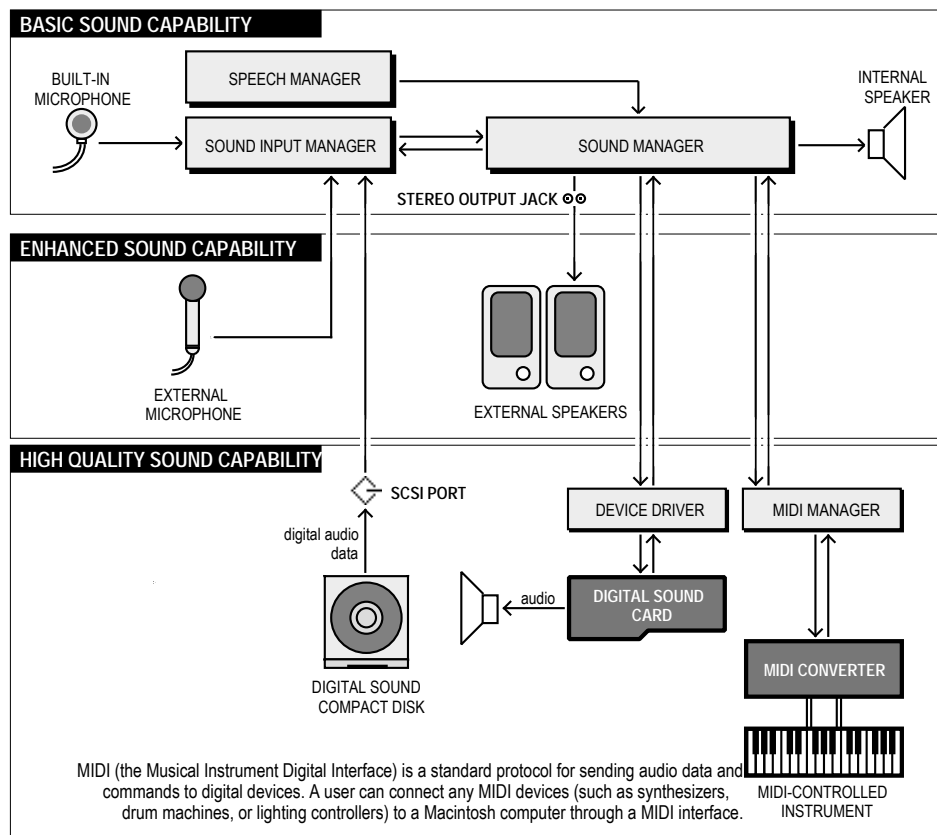


FIG 1 - SOUND CAPABILITIES OF MACINTOSH COMPUTERS

Sound Data

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

- **Square Wave Data.** Square-wave data can be used to play a simple sequence of sounds in which each sound is described by frequency (pitch), amplitude (volume), duration.
- **Wave-Table Data.** Wave table data may be used to produce more complex sounds than are possible using square-wave data. A wave cycle is represented as an array of bytes that describe the timbre (tone) of a sound at a point in the cycle.
- **Sampled-Sound Data.** Sampled sound data is 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 Sound Manager supports 16-bit stereo audio samples with sample rates up to 64kHz.

Storing Sampled Sounds

Sampled-sound data comprises a series of **sample frames**. 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 uses **sound components** to modify sound data. A sound component is a stand-alone code resource that can perform operations on sound data such as compression, decompression, and converting sample rates. Sound components may be hooked together in series to perform complex tasks, as shown in the example at Fig 2.

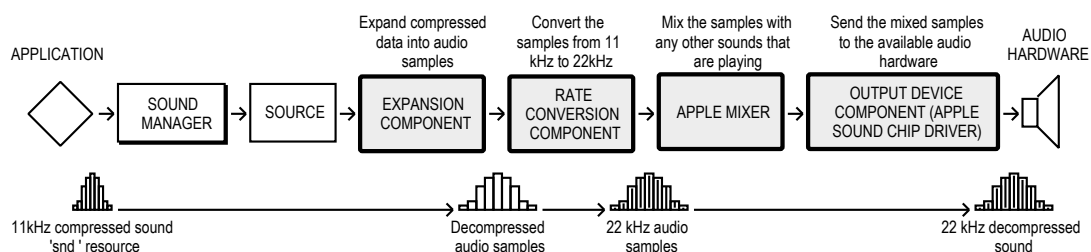


FIG 2 - A TYPICAL SOUND COMPONENT CHAIN

The Sound Manager is aware of the sound output device selected by the user and assembles a component chain suitable for producing the desired quality of sound on that device. Thus your application is generally unaware of the sound component chain assembled to produce a sound on the selected output device.

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

Sound Resources

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

Sound Production

Sound Channels

A Macintosh produces sound when the Sound Manager sends data through a **sound channel** to the audio hardware. A sound channel is basically a queue of **sound commands** (see below), which might be placed into the sound channel by your application or by the Sound Manager itself.

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

```
struct SndChannel
{
    SndChannelPtr  nextChan;    // Pointer to next channel.
    Ptr            firstMod;    // (Used internally.)
    SndCallbackUPP callBack;    // Pointer to callback function.
    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[128];  // (Used internally.)
}
typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

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 or modify sound, or otherwise contribute to the overall sound production process. 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;
```

¹ A 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.

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.
soundCmd  = 80  Install a sampled sound as a voice in a channel.
bufferCmd = 81  Play a buffer of sampled-sound data.
```

Carbon Note

Several Sound Manager sound commands are not available in Carbon.

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.

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,"My 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 80 80 80 80 81 82 82"
    "$82 83 82 82 81 80 80 7F 7F 7E 7D 7D 7C 7C"
    (Rest of sampled sound data.)
};
```

Note that the sound resource header section indicates that the sound is defined using sampled-sound data. Note also that the sound commands section contains a call to a single sound command (the `bufferCmd` command (0x51)) and that 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**², which immediately follows the sound commands section.

Note that the first part of the sampled sound header contains information about the sample and that the sampled sound data is itself part of the sampled sound header.

² 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 into a sound channel one at a time by calling `SndDoCommand` or you can bypass a sound queue altogether by calling the `SndDoImmediate`.

Synchronous and Asynchronous Sound

You can play sounds either **synchronously** or **asynchronously**. When your application plays a sound synchronously, it cannot continue executing until the sound has finished playing. When your application plays a sound asynchronously, it 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

Carbon Note

The Sound Manager function `SndStartFilePlay` (starts a file playing from disk), together with the associated functions `SndPauseFilePlay`, `SndStopFilePlay`, `SndPlayDoubleBuffer`, are not available in Carbon.

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 the sound data is compressed, `SndPlay` decompresses the data in order to play the sound.

Channel Allocation

If you pass `NULL` in the first parameter of `SndPlay`, a sound channel will be automatically allocated to play the sound and then automatically disposed of when the sound has finished playing.

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`, 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 — Mac OS 8/9

On Mac OS 8/9, the Sound Input Manager provides a high-level function that allow your application to record sounds from the user and store them in memory. When you call this functions, the Sound Input Manager presents the sound recording dialog shown at Fig 3.



FIG 3 - SOUND RECORDING DIALOG — MAC OS 8/9

Carbon Note

The Sound Manager function `SndRecordToFile` (records sound data to a file) is not available in Carbon.

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 in 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` in 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 Quality

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

Constant	Value	Meaning
<code>siCDQuality</code>	'cd'	44.1kHz, stereo, 16 bit.
<code>siBestQuality</code>	'best'	22kHz, mono, 8 bit.
<code>siBetterQuality</code>	'betr'	22kHz, mono, 3:1 compression.
<code>siGoodQuality</code>	'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 same fidelity as a commercial compact disc occupies about 5.3 MB of disk space, and one minute of telephone-quality speech takes up more than half a megabyte.

Speech

The Speech Manager converts text into sound data and passes to the Sound Manager. The Speech Manager utilises a **speech synthesiser**, which 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, produces the speech on that channel, 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, you are free to release the memory allocated for the string as soon as `SpeakString` returns, the reason being that `SpeakString` copies the string into an internal buffer,.

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.

Relevant Constants, Data Types, and Functions

Constants

Recording Qualities

siCDQuality = FOUR_CHAR_CODE('cd ') 44.1kHz, stereo, 16 bit.
siBestQuality = FOUR_CHAR_CODE('best') 22kHz, mono, 8 bit.
siBetterQuality = FOUR_CHAR_CODE('betr') 22kHz, mono, MACE 3:1.
siGoodQuality = FOUR_CHAR_CODE('good') 22kHz, mono, MACE 6:1.

Typical Sound Commands

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.
soundCmd = 80 Install a sampled sound as a voice in a channel.
bufferCmd = 81 Play a buffer of sampled-sound data.

Data Types

Sound Channel Structure

```
struct SndChannel
{
    SndChannelPtr nextChan;    // Pointer to next channel.
    Ptr firstMod;             // (Used internally.)
    SndCallbackUPP callBack;  // Pointer to callback function.
    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[128];    // (Used internally.)
}
typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

Sound Command Structure

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

Functions

Playing Sound Resources

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

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);
```

Generating Speech

```
OSErr  SpeakString(ConstStr255Param textToBeSpoken);  
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 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:

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 ASStructure currently in use?

```
OSErr  AS_Initialise(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 ASStructures equal to the requested number of sound channels.

OSErr AS_PlayID(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, 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).

OSErr AS_PlayHandle(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 AS_PlayHandle function is similar to AS_PlayID, except that it supports a special case: You can pass AS_PlayHandle a NULL 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).

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

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

Returns: 0 No errors.
 1 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 AsyncSoundLib 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 NULL 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.

```
void AS_CloseChannel (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 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.

```
void AS_CloseDown(void);
```

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 SoundAndSpeech Listing

```
// *****
// SoundAndSpeech.c CARBON EVENT MODEL
// *****
//
// This program opens a modeless dialog containing five 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.
//   • Record sound resource (Mac OS 8/9 only).
//   • Speak text string.
//
// • Asynchronous group:
//   • Play sound resource.
//   • Speak text string.
//
// The asynchronous sound sections of the program utilise a special library called
// AsyncSoundLibPPC, which must be included in the CodeWarrior project.
//
// The program utilises the following resources:
//
// • A 'plst' resource.
//
// • A 'DLOG' resource and associated 'DITL', 'dlgx', and 'dftb' resources (all purgeable).
//
// • 'CNTL' resources (purgeable) for the controls within the dialog.
//
// • Two 'snd ' resources, one for synchronous playback (purgeable) and one for 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.
//
// • Two 'STR#' resources containing "speak text" strings and error message strings (all
//   purgeable).
//
// • 'hrct' and 'hwin' resources (purgeable) for balloon help.
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//   doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// Each time it is invoked, the function doRecordResource creates a new 'snd' resource with a
// unique ID in the resource fork of a file titled "SoundResources".
//
// *****

// ..... includes

#include <Carbon.h>
#include <string.h>

// ..... defines

#define rDialog          128
#define iDone            1
#define iPlayResourceSync 4
#define iRecordResource  5
#define iSpeakTextSync   6
#define iPlayResourceASync 7
```

```

#define iSpeakTextAsync      8
#define rPlaySoundResourceSync 8192
#define rPlaySoundResourceAsync 8193
#define rSpeechStrings      128
#define rErrorStrings       129
#define eOpenDialogFail     1
#define eCannotInitialise   2
#define eGetResource        3
#define eMemory             4
#define eMakeFSSpec         5
#define eWriteResource      6
#define eNoChannelsAvailable 7
#define ePlaySound          8
#define eSndPlay            9
#define eSndRecord         10
#define eSpeakString        11
#define rColourIcon1        128
#define rColourIcon2        129
#define kMaxChannels        8
#define kOutOfChannels      1

// ..... global variables

Boolean    gRunningOnX = false;
DialogRef  gDialogRef;
CIconHandle gColourIconHdl1;
CIconHandle gColourIconHdl2;

// ..... AsyncSoundLib attention flag

Boolean    gCallAS_CloseChannel = false;

// ..... function prototypes

void      main                (void);
void      doPreliminaries     (void);
OSStatus  windowEventHandler  (EventHandlerCallRef,EventRef,void *);
void      doIdle              (void);
void      doInitialiseSoundLib (void);
void      doDialogHit         (SInt16);
void      doPlayResourceSync   (void);
void      doRecordResource     (void);
void      doSpeakStringSync    (void);
void      doPlayResourceAsync  (void);
void      doSpeakStringAsync   (void);
void      doSetUpDialog        (void);
void      doErrorAlert         (SInt16);
void      helpTags             (DialogRef);

// ..... AsyncSoundLib function prototypes

OSErr AS_Initialise  (Boolean *,SInt16);
OSErr AS_GetChannel  (SInt32,SndChannelPtr *);
OSErr AS_PlayID      (SInt16, SInt32 *);
OSErr AS_PlayHandle  (Handle,SInt32 *);
void  AS_CloseChannel (void);
void  AS_CloseDown   (void);

// ***** main

void main(void)
{
    SInt32      response;
    EventTypeSpec windowEvents[] = { { kEventClassWindow, kEventWindowClose },
                                      { kEventClassMouse, kEventMouseDown } };

    // ..... do preliminaries

    doPreliminaries();

```

```

// ..... disable Quit item in Mac OS X Application menu
DisableMenuCommand(NULL,'quit');

// ..... install a timer
InstallEventLoopTimer(GetCurrentEventLoop(),0,TicksToEventTime(10),
                      NewEventLoopTimerUPP((EventLoopTimerProcPtr) doIdle),NULL,
                      NULL);

// ..... open and set up dialog
if(!(gDialogRef = GetNewDialog(rDialog,NULL,(WindowRef) -1)))
{
    doErrorAlert(eOpenDialogFail);
    ExitToShell();
}

SetPortDialogPort(gDialogRef);
SetDialogDefaultItem(gDialogRef,kStdOkItemIndex);

ChangeWindowAttributes(GetDialogWindow(gDialogRef),kWindowStandardHandlerAttribute |
                      kWindowCloseBoxAttribute,
                      kWindowCollapseBoxAttribute);

InstallWindowEventHandler(GetDialogWindow(gDialogRef),
                          NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler),
                          GetEventTypeCount(windowEvents),windowEvents,0,NULL);

Gestalt(gestaltMenuMgrAttr,&response);
if(response & gestaltMenuMgrAquaLayoutMask)
{
    helpTags(gDialogRef);
    gRunningOnX = true;
}

doSetUpDialog();

// ..... initialise AsyncSoundLib
doInitialiseSoundLib();

// ..... get colour icons
gColourIconHdl1 = GetCIcon(rColourIcon1);
gColourIconHdl2 = GetCIcon(rColourIcon2);

// ..... run application event loop
RunApplicationEventLoop();
}

// ***** doPreliminaries

void doPreliminaries(void)
{
    MoreMasterPointers(64);
    InitCursor();
    FlushEvents(everyEvent,0);
}

// ***** windowEventHandler

OSStatus windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                           void* userData)
{
    OSStatus    result = eventNotHandledErr;
    UInt32      eventClass;

```



```

UInt32      eventKind;
EventRecord eventRecord;
SInt16      itemHit;

eventClass = GetEventClass(eventRef);
eventKind  = GetEventKind(eventRef);

switch(eventClass)
{
    case kEventClassWindow:                                // event class window
        switch(eventKind)
        {
            case kEventWindowClose:
                AS_CloseDown();
                QuitApplicationEventLoop();
                result = noErr;
                break;
        }

    case kEventClassMouse:                                // event class mouse
        ConvertEventRefToEventRecord(eventRef,&eventRecord);
        switch(eventKind)
        {
            case kEventMouseDown:
                if(IsDialogEvent(&eventRecord))
                {
                    if(DialogSelect(&eventRecord,&gDialogRef,&itemHit))
                        doDialogHit(itemHit);
                    result = noErr;
                }
                break;
        }
        break;
}

return result;
}

// ***** doIdle

void doIdle(void)
{
    Rect      theRect, eraseRect;
    UInt32    finalTicks;
    SInt16    fontNum;
    static Boolean flip;

    SetRect(&theRect,262,169,294,201);
    SetRect(&eraseRect,310,170,481,200);

    if(gCallAS_CloseChannel)
    {
        AS_CloseChannel();

        GetFNum("\pGeneva",&fontNum);
        TextFont(fontNum);
        TextSize(10);
        MoveTo(341,189);
        DrawString("\pAS_CloseChannel called");
        QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
        Delay(45,&finalTicks);
    }

    if(flip)
        PlotCIcon(&theRect,gColourIconHdl1);
    else
        PlotCIcon(&theRect,gColourIconHdl2);

    flip = !flip;
}

```

```

    EraseRect(&eraseRect);
}

// ***** doInitialiseSoundLib

void doInitialiseSoundLib(void)
{
    if(AS_Initialise(&gCallAS_CloseChannel,kMaxChannels) != noErr)
    {
        doErrorAlert(eCannotInitialise);
        ExitToShell();
    }
}

// ***** doDialogHit

void doDialogHit(SInt16 item)
{
    switch(item)
    {
        case iDone:
            AS_CloseDown();
            QuitApplicationEventLoop();
            break;

        case iPlayResourceSync:
            doPlayResourceSync();
            break;

        case iRecordResource:
            doRecordResource();
            break;

        case iSpeakTextSync:
            doSpeakStringSync();
            break;

        case iPlayResourceASync:
            doPlayResourceASync();
            break;

        case iSpeakTextAsync:
            doSpeakStringAsync();
            break;
    }
}

// ***** doPlayResourceSync

void doPlayResourceSync(void)
{
    SndListHandle sndListHdl;
    SInt16        resErr;
    OSErr         osErr;
    ControlRef    controlRef;

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

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

```

```

        ReleaseResource((Handle) sndListHdl);

        GetDialogItemAsControl(gDialogRef,iPlayResourceSync,&controlRef);
        SetControlValue(controlRef,0);
    }
}

// ***** doRecordResource

void doRecordResource(void)
{
    SInt16      oldResFileRefNum, theResourceID, resErr, tempResFileRefNum;
    BitMap      screenBits;
    Point       topLeft;
    OSErr       memErr, osErr;
    Handle       soundHdl;
    FSSpec       fileSpecTemp;
    ControlRef   controlRef;

    oldResFileRefNum = CurResFile();

    GetQDGlobalsScreenBits(&screenBits);
    topLeft.h = (screenBits.bounds.right / 2) - 156;
    topLeft.v = 150;

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

    osErr = FSMakeFSSpec(0,0,"\\pSoundResources",&fileSpecTemp);
    if(osErr == noErr)
    {
        tempResFileRefNum = FSpOpenResFile(&fileSpecTemp,fsWrPerm);
        UseResFile(tempResFileRefNum);
    }
    else
        doErrorAlert(eMakeFSSpec);

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

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

    CloseResFile(tempResFileRefNum);
}

DisposeHandle(soundHdl);
UseResFile(oldResFileRefNum);

```

```

    GetDlgItemAsControl(gDialogRef,iRecordResource,&controlRef);
    SetControlValue(controlRef,0);
}

// ***** doSpeakStringSync

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

    activeChannels = SpeechBusy();

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

    osErr = SpeakString(theString);
    if(osErr != noErr)
        doErrorAlert(eSpeakString);

    while(SpeechBusy() != activeChannels)
        ;

    GetDlgItemAsControl(gDialogRef,iSpeakTextSync,&controlRef);
    SetControlValue(controlRef,0);
}

// ***** doPlayResourceASync

void doPlayResourceASync(void)
{
    SInt16 error;

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

// ***** doSpeakStringAsync

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

    GetIndString(theString,rSpeechStrings,2);
    resErr = ResError();
    if(resErr != noErr)
    {
        doErrorAlert(eGetResource);
        return;
    }

    osErr = SpeakString(theString);
    if(osErr != noErr)
        doErrorAlert(eSpeakString);
}

// ***** doSetUpDialog

```

```

void doSetUpDialog(void)
{
    SInt16          a;
    Point           offset;
    ControlRef       controlRef;
    ControlButtonGraphicAlignment alignConstant = kControlBevelButtonAlignLeft;
    ControlButtonTextPlacement placeConstant = kControlBevelButtonPlaceToRightOfGraphic;

    offset.v = 1;
    offset.h = 5;

    for(a=iPlayResourceSync;a<iSpeakTextAsync+1;a++)
    {
        GetDialogItemAsControl(gDialogRef,a,&controlRef);
        SetControlData(controlRef,kControlEntireControl,kControlBevelButtonGraphicAlignTag,
            sizeof(alignConstant),&alignConstant);
        SetControlData(controlRef,kControlEntireControl,kControlBevelButtonGraphicOffsetTag,
            sizeof(offset),&offset);
        SetControlData(controlRef,kControlEntireControl,kControlBevelButtonTextPlaceTag,
            sizeof(placeConstant),&placeConstant);
    }

    if(gRunningOnX)
    {
        GetDialogItemAsControl(gDialogRef,iRecordResource,&controlRef);
        DeactivateControl(controlRef);
    }
}

// ***** doErrorAlert

void doErrorAlert(SInt16 errorStringIndex)
{
    Str255 errorString;
    SInt16 itemHit;

    GetIndString(errorString,rErrorStrings,errorStringIndex);
    StandardAlert(kAlertCautionAlert,errorString,NULL,NULL,&itemHit);
}

// ***** helpTags

void helpTags(DialogRef dialogRef)
{
    HMHelpContentRec helpContent;
    SInt16          a;
    ControlRef       controlRef;

    memset(&helpContent,0,sizeof(helpContent));
    HMSetTagDelay(500);
    HMSetHelpTagsDisplayed(true);

    helpContent.version = kMacHelpVersion;
    helpContent.tagSide = kHMOutsideTopCenterAligned;
    helpContent.content[kHMMinimumContentIndex].contentType = kHMStringResContent;
    helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmResID = 130;

    for(a = 1;a <= 5; a++)
    {
        if(a == 2)
            continue;
        helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = a;
        GetDialogItemAsControl(dialogRef,a + 3,&controlRef);
        HMSetControlHelpContent(controlRef,&helpContent);
    }
}

// *****

```

Demonstration Program SoundAndSpeech Comments

When this program is run, the user should click on the various buttons in the dialog to play back and record (Mac OS 8/9 only) sound resources 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 "working man" icon 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 file titled "SoundResources".

On Mac OS 9, ensure that the Speech Manager extension is activated before running this program.

defines

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.

main

A timer is installed and set to fire repeatedly every 10 ticks. When the timer fires, the function doIdle is called.

doInitialiseSoundLib is called to initialise the AsynchSoundLib library.

doidle

doIdle is called every time the timer fires.

The "attention" flag (gAS_CloseChannel) 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 "working man" icon to indicate to the user that AS_CloseChannel has just been called.

The next block draws one or other of the two "working man" icons, following which the interior of the group box is erased.

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.

doPlayResourceSync

doPlayResourceSync 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 39940-byte 'snd ' resource being used contains one command only (bufferCmd). The compressed sound header indicates MACE 3:1 compression. The sound length is 119568 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.

doRecordResource

On Mac OS 8/9 only, doRecordResource uses SndRecord to record a sound synchronously and then saves the sound in a 'snd ' resource. The 'snd ' resource will be saved to the resource fork of the file "SoundResources".

The first line saves the file reference number of the current resource file. The next three 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.

The next block opens the resource fork of the file "SoundResources" and makes it the current resource file.

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 after closing the resource fork of the file, disposing of the relocatable block, and restoring the saved resource file reference number.

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 resource fork of the current resource file, the do/while loop first finds 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 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.

The relocatable block is then disposed of, the resource fork of the file "SoundResources" is closed, and the saved resource file reference number is restored.

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.

doPlayResourceASync

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

Note: The 24194-byte 'snd ' resource being used contains one command only (bufferCmd). The compressed sound header indicates no compression. The sound length is 24195 frames. The 8-bit mono sound was sampled at 5kHz.

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 NULL 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 message 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.

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

Within doSetUpDialog, the Record Sound Resource bevel button is disabled if the program is running on OS X.