Inside Macintosh: ***Sound*** *addendum*

Introduction

The Sound Manager has been valuable since the introduction of the Macintosh II in 1987. The Sound Manager is Apple's digital audio software solution that allows any application to play and record sounds using the sound hardware found on Macintosh computers. Sound Manager 3.0, released in 1993, added support for 16-bit CD-quality audio, redirection of sound to third-party hardware cards, and plug-in audio compression/decompression software (codecs).

Releases

The following is a brief history of the Sound Manager releases following the original 3.0 release. These features were not documented in the original *Inside Macintosh: Sound* book, which documented the 3.0 release.

Sound Manager 3.1 - July 1995

Sound Manager 3.1 adds two new audio codecs, significant performance increases on the Power Macintosh line of computers, and asynchronous alert sounds. It is completely backwards compatible with previous versions of the Sound Manager.

The increased performance on Power Macintosh is one noticeable feature of Sound Manager 3.1. Here's a table that shows the real time usage of a Power Macintosh 6100/60 under various test cases playing a single sound. This compares the emulated 68k code of the previous Sound Manager with the new PowerPC code of Sound Manager 3.1.

	Emulated 68k	Native PowerPC
8-bit, mono	5%	1.6%
8-bit, mono, w/sample rate conversion	12%	2.2%
16-bit, stereo	13%	1.9%
16-bit, stereo, w/sample rate conversion	26%	2.3%
16-bit, stereo, w/sample rate conversion, IMA	41%	6.6%

As you can see, the new Sound Manager performs 3-6 times faster on your PowerMac. In addition, recording with the Sound Input Manager has also been enhanced, and will be 7-9 times faster.

Native PowerPC code

All of the critical sound components are native PowerPC code. This includes the mixer, sample rate converter, format converters, MACE 3:1 and 6:1 compression and decompressers. The sound input interrupt handler also uses native PowerPC code to boost performance during recording. These enhancements on Power Macintosh will let games and QuickTime movies play more smoothly and provide higher capture rates when recording digital video.

Three new audio codecs: IMA, µLaw, and "little-endian"

The IMA 4:1 audio compression format is based on a standard proposed by the Interactive Multimedia Association, and is used to compress 16-bit sound with a ratio of 4:1. It is particularly good at compressing CD-quality music and is fully integrated into QuickTime.

The µLaw 2:1 format (pronounced "mu-law") is an international standard for compressing voice-quality audio (typically 16-bit, 8 kHz speech) with a ratio of 2:1. It is often used in telephony applications, and also on the Internet as the encoding format for ".au" sound files.

The "little-endian" codec provides support for the byte-ordering used on Intel-standard personal computers. This codec lets QuickTime directly play 16-bit sound files encoded in the popular ".wav" file format.

Asynchronous Alert Sounds

The Sound Manager previously tied up your Macintosh while playing an alert sound, forcing you to wait until the sound was done playing before you could continue. Sound Manager 3.1 removes this limitation by playing alert sounds asynchronously, so alert dialogs and other interface elements can continue processing while the alert sound is playing.

New sounds are queued within the same process and additional sounds from other processes are mixed together. This gives the user a perceived performance increase. You'll probably feel that the machine is running faster when a system alert sound doesn't lock up your computer anymore. You can disable this (but why would you want to?) by setting the sysBeepSynchronous flag when calling SndSetSysBeepState().

New Sound Commands

rateMultiplierCmd
getRateMultiplierCmd

New Routines

SndGetInfo()
SndSetInfo()

Sound Manager 3.2 - March 1996

These new features are only available with Sound Manager version 3.2 or later. Check for this by calling SndSoundManagerVersion() for the installed version.

New Routines

GetSoundOutputInfo()
SetSoundOutputInfo()
ParseAIFFHeader()
ParseSndHeader()
GetCompressionName()

Sound Manager 3.2.1 - August 1996

These new feature are only available with Sound Manager version 3.2.1 or later. Check for this by calling SndSoundManagerVersion() for the installed version.

Pre-mixer Effects

The siPreMixerSoundComponent has been added for installing a pre-mixer sound component to provide specialized audio effects, such as the Sound Sprockets sound localizer. Refer to the new Sprockets Games documentation for use with this new sound effect. You can find more information about the Sound Sprockets component at the World Wide Web page http://devworld.apple.com/dev/games/

Sound Clock

There are two new sound commands, clockComponentCmd and getClockComponentCmd. These were added to support a sound clock which is used by QuickTime as the main time base. The Sound Manager sound clock is only available when QuickTime is installed. This keeps QuickTime's video in synch with the audio. There is one sound clock per source (e.g. sound channel) which is updated with each hardware buffer.

Determining which Sound Manager is installed

The Sound Manager provides a routine, SndSoundManagerVersion(), which returns the currently installed version. To check for Sound Manager 3.1 or later, use the following code:

```
Boolean HasSoundManager3_1(void)
{
	NumVersion   version;

	version = SndSoundManagerVersion();
	if (   (version.majorRev > 3) ||
		((version.majorRev == 3) && (version.minorAndBugRev >= 0x10)))
		return (true);
	else
		return (false);
}
```

Sound Commands

The following are new sound commands added since the release of Sound Manager 3.0 as documented in *Inside Macintosh: Sound*.

clockComponentCmd

Use the clockComponentCmd to turn the clock on or off. Set the param1 as a Boolean, using true to turn it on and false to turn it off.

```
SndCommand          cmd;

cmd.cmd = clockComponentCmd;
cmd.param1 = true;    // turn the clock on
err = SndDoImmediate(chan, &clockComponentCmd);
```

getClockComponentCmd

Use the getClockComponentCmd to obtain the ComponentInstance of the clock, setting the param2 field as the address of your ComponentInstance variable.

```
SndCommand          cmd;
ComponentInstance   clock;

cmd.cmd = getClockComponentCmd;
cmd.param2 = (long)&clock;   // get the clock component
err = SndDoImmediate (chan, &getClockComponentCmd);
```

rateMultiplierCmd

The rateMultiplierCmd uses a fixed-point value to provide a multiplier to the playback rate of all sounds played on this channel. This allows you to vary the sample rate of the sound being played, and thus control its pitch. The getRateMultiplierCmd returns the current rate multiplier. For example, to play all sounds on a channel shifted up one octave in pitch, you could use the following code:

```
OSErr RaisePitchOneOctave(SndChannelPtr chan)
{
       SndCommand    cmd;
       OSErr         err;

       cmd.cmd = rateMultiplierCmd;
       cmd.param1 = 0;
       cmd.param2 = 0x00020000;    // rate of 2.0

       err = SndDoImmediate(chan, &cmd);
       return (err);
}
```

The rateMultiplierCmd is more useful than the old rateCmd, which applied only to the sound currently playing and was based on the sampling rate of the hardware.

Sound Information Selectors

The following are new sound informational selectors added since the release of Sound Manager 3.0 as documented in *Inside Macintosh: Sound*. These selectors are used to obtain information and control the sound environment using the GetSoundOutputInfo(), SetSoundOutputInfo(), SndGetInfo(), SndSetInfo(), or SPBGetDeviceInfo() and SPBSetDeviceInfo() routines.

siHardwareBusy

Used by all input and output devices. This returns the state of the hardware. Typically, this means whether or not the hardware interrupts are active. For input devices, the infoData parameter points to a short word. For output devices, the infoPtr is a short word containing the value. A value of 0 represents that the hardware is not busy, whereas 1 represents busy. This selector is only supported by the get calls, since it doesn't make since to "set" a device busy.

```
short    hwBusy;

err = GetSoundOutputInfo(nil, siHardwareBusy, &hwBusy);
```

siHardwareFormat

Added the siHardwareFormat for output devices. Use this selector with the GetSoundOutputInfo() and SndGetInfo() routines. This returns a SoundComponentData structure of the format used by the output hardware device.

```
SoundComponentData    outputFormat;

err = GetSoundOutputInfo(nil, siHardwareFormat, &outputFormat);
```

siHardwareMute

This output selector was previously defined, but should be documented and implemented as returning the mute state of all sources that can be heard. For example, the PowerMac has a speaker and headphone source. If no headphones are plugged in, then return the mute state of the speakers. If headphones are inserted, then return the mute state of both (e.g. siHardwareMute is muted when both speakers and headphones are muted). The idea is if the user has sources that can be heard and all are muted, then the hardware has been muted. If any one of the sources can be heard, then the hardware is not muted.

```
short    hwMuted;

err = GetSoundOutputInfo(nil, siHardwareMute, &hwMuted);
```

siHardwareVolume

This output selector was previously defined, but should be documented and implemented as returning the volume of the device that can be heard. For example, the PowerMac can have the speaker and the headphones both be audible to the user. In this case, return the loudest volume setting. If both the speaker and the headphones are muted, then again return the loudest setting. If one is muted and the other is audible, then return the volume of the audible source.

```
unsigned long    hwVolume;

err = GetSoundOutputInfo(nil, siHardwareVolume, &hwVolume);
```

siPreMixerSoundComponent

Used to install a given sound component into the component chain. The given component will be installed just before the mixer component. This sound component can access the audio stream and modify the data prior to its being sent to the mixer. An example of its use is shown below by installing the Sound Sprockets sound localizer.

```
SoundComponentLink  soundLink;

soundLink.description.componentType = kSoundEffectsType;
soundLink.description.componentSubType = kSSpLocalizationSubType;
soundLink.description.componentManufacturer = kAnyComponentManufacturer;
soundLink.description.componentFlags = 0;
soundLink.description.componentFlagsMask = kAnyComponentFlagsMask;
soundLink.mixerID = nil;
soundLink.linkID = nil;

err = SndSetInfo(chan, siPreMixerSoundComponent, &soundLink);

       // configure the localization settings
       // and send them to the sound component

err = SndSetInfo(chan, siSSpLocalization, &localizationSettings);
```

siSetupCDAudio

This selector is only used by input devices that have special requirements in order to hear the audio from the CD player. For example, currently the 840AV and PowerMacs require the user to open the Sound control panel, select the Inputs panel, and then open the Options dialog. From here they have to select the CD Input and also check the Play-Through option. The new siSetupCDAudio avoids this troublesome operation by setting up the input device to allow you to hear audio CDs from the audio CD player.

The infoData parameter points to a short word. A value of 1 means to setup the hardware (e.g. set input source to CD and turn on Play-Through). A value of 0 means to return the input hardware to some initial state, either the default settings or the settings prior to setting up for the CD audio. Any other value is an error. If the input device had no special needs for CD audio (e.g. the audio is heard regardless of the input hardware settings) then the selector is not supported and it returns an error. The SPBGetDeviceInfo() should return a 1 or 0 value depending on its current setting. If the input source is the CD and Play-Through, if required, is on, then return the value of 1. This should happen even if nothing ever requested setting up for CD audio. It should return the state of the actual condition of the input hardware.

Sound Manager Routines

The following are new Sound Manager routines added since the release of Sound Manager 3.0 as documented in *Inside Macintosh: Sound*.

SndGetInfo() and SndSetInfo()

```
extern pascal OSErr SndGetInfo(SndChannelPtr chan, OSType selector, void
*infoPtr);

extern pascal OSErr SndSetInfo(SndChannelPtr chan, OSType selector, const void
*infoPtr);
```

The two new routines SndGetInfo() and SndSetInfo() are used to get and set information about the sound environment. Both routines use a selector based interface similar to the SPBGetDeviceInfo() and SPBSetDeviceInfo() routines found in the Sound Input Manager, and in fact they use the same sound information selectors.

SndGetInfo() and SndSetInfo() operate on an open Sound Manager channel, and can be used to retrieve and change information about the channel, including hardware settings. These routines should be used instead of attempting to communicate directly with sound components.

These new calls are only available with Sound Manager version 3.1 or later. Check for this by calling SndSoundManagerVersion() for the installed version. Note that you can always open a sound channel for the hardware device that you desire by passing kUseOptionalOutputDevice as the synth parameter and the component reference as the init parameter.

```
OSErr OpenChannel(OSType myType)
{
    ComponentDescription    searching;
    Component               outputDevice;
    OSErr                   err;

    // search for a sound output device component
    searching.componentType = kSoundOutputDeviceType;
    searching.componentSubType = myType;
    searching.componentManufacturer = kAnyComponentManufacturer;
    searching.componentFlags = 0;
    searching.componentFlagsMask = kAnyComponentFlagsMask;
    outputDevice = nil;
    outputDevice = FindNextComponent(outputDevice, &searching);

    if (outputDevice == nil)
        err = cantFindHandler;          /*component not found*/
    else
    {
        gChan = nil;
        err = SndNewChannel(&gChan, kUseOptionalOutputDevice,
                            (long)outputDevice, nil);
    }
    return (err);
}
```

For example, to determine the current hardware sampling rate of the given sound channel you could use this code:

```
UnsignedFixed        sampleRate;

err = SndGetInfo(gChan, siSampleRate, &sampleRate);
```

GetSoundOutputInfo() and SetSoundOutputInfo()

```
pascal OSErr GetSoundOutputInfo(Component outputDevice, OSType selector,
                                    void *infoPtr);

pascal OSErr SetSoundOutputInfo(Component outputDevice, OSType selector,
                                    const void *infoPtr);
```

These two routines get and set information about the sound environment: GetSoundOutputInfo() and SetSoundOutputInfo(). Both routines use a selector based interface similar to the SPBGetDeviceInfo() and SPBSetDeviceInfo() routines found in the Sound Input Manager, and in fact they use the same sound info selectors.

GetSoundOutputInfo() and SetSoundOutputInfo() operate directly on a sound output device, and can be used to retrieve and change information about the hardware settings. These routines should be used instead of attempting to communicate directly with sound output components. Setting the output device parameter to nil causes the default output device to be used. These calls are similar to GetSndInfo() and SetSndInfo() but do not require an opened sound channel. For example, to determine the sampling rate of the sound hardware on the default output device, you could use this code:

```
OSErr GetCurrentSampleRate(UnsignedFixed *sampleRate)
{
    OSErr           err;

    err = GetSoundOutputInfo(nil, siSampleRate, sampleRate);
    return (err);
}
```

ParseAIFFHeader()

```
pascal OSErr ParseAIFFHeader(short fRefNum, SoundComponentData *sndInfo,
                      unsigned long *numFrames, unsigned long *dataOffset)
```

ParseAIFFHeader() returns information describing the audio data in the given AIFF file. The fRefNum parameter specifies the open AIFF file to use. The sndInfo parameter is a SoundComponentData structure that returns the following information about the format of the sound in the AIFF file:

```
    flags           - always returns 0
    format          - the sound format (i.e. 'raw ', 'twos', 'MAC3', etc.)
    numChannels     - the number of channels  (i.e. 1 = mono, 2 = stereo)
    sampleSize      - the sample size (i.e. 8 = 8-bit, 16 = 16-bit)
    sampleRate      - the sampling rate (in samples/second)
    sampleCount     - the number of audio samples in the file
    buffer          - always returns 0
    reserved        - always returns 0
```

The numFrames parameter returns the number of frames of audio data in the file, and the dataOffset parameter returns the byte offset of the first audio sample in the file.

ParseSndHeader()

```
pascal OSErr ParseSndHeader(SndListHandle sndHandle,
                    SoundComponentData *sndInfo, unsigned long *numFrames,
                    unsigned long *dataOffset)
```

ParseSndHeader() returns information describing the audio data in the given 'snd ' resource handle. The sndHandle parameter specifies the sound handle to use. The sndInfo parameter is a SoundComponentData structure that returns the following information about the format of the sound in the handle:

```
flags           - always returns 0
format          - the sound format (i.e. 'raw ', 'twos', 'MAC3', etc.)
numChannels     - the number of channels  (i.e. 1 = mono, 2 = stereo)
sampleSize      - the sample size (i.e. 8 = 8-bit, 16 = 16-bit)
sampleRate      - the sampling rate (in samples/second)
sampleCount     - the number of audio samples in the handle
buffer          - always returns 0
reserved        - always returns 0
```

The numFrames parameter returns the number of frames of audio data in the handle, and the dataOffset parameter returns the byte offset of the first audio sample in the handle.

GetCompressionInfo()

```
pascal OSErr GetCompressionInfo(short compressionID, OSType format,
              short numChannels, short sampleSize, CompressionInfoPtr cp)
```

For a given AIFF file or snd resource, the information contained within it might be used to determine basic characteristics of the sound such as its duration.

```
duration = numSampleFrames / sampleRate
```

Note that this is a valid calculation for an uncompressed sound. But this calculation returns an incorrect result for a compressed sound. The problem here is that each sample frame in a compressed sound is composed of one or more *packets* rather than sample points (see *Inside Macintosh: Sound* page 2-9 to 2-11), and each packet in that compressed sound can itself represent *several* sample points. We therefore need a way to determine the number of samples in a packet in order to get an accurate calculation.

The compressionID parameter defines the compression algorithm used on the sample. The AIFF-C Extended Common Chunk does not contain a compressionID field. In this case (and when using snd resources where the OSType describing the compression format is known) you should always pass the constant fixedCompression in this parameter and the OSType in the format parameter. The format field will then contain the OSType representing the compression format. If you set the compressionID field in a compressed sound header to any value other than fixedCompression, then the format field is set to zero. The format parameter is the OSType describing the format of the compressed sound data. If you pass the constant fixedCompression in the compressionID parameter you will need to pass a valid compression type here. Some of the valid format types are:

```
NONE                - sound is not compressed
MAC3                - compression format is MACE 3:1
MAC6                - compression format is MACE 6:1
ima4                - compression format is IMA 4:1
```

There are some snd resources that do not store an OSType in the format field of the compressed sound header describing the compression format. You can still use GetCompressionInfo() in this case by passing in the compressionID and passing 0 in the format parameter. The correct OSType will be returned in the format field of the CompressionInfo structure. Using the appropriate fields from an AIFF-C Extended Common Chunk or our snd resource compressed sound header, we can make the call to GetCompressionInfo():

example Extended Common Chunk

```
myExtendedCommonChunk.ckID = 'COMM';
myExtendedCommonChunk.ckSize = 34;
myExtendedCommonChunk.numChannels = 1;
myExtendedCommonChunk.numSampleFrames = 7633;
myExtendedCommonChunk.sampleSize = 8;
myExtendedCommonChunk.sampleRate = 22254.54545;
myExtendedCommonChunk.compressionType = MAC3;
myExtendedCommonChunk.compressionName = “MACE 3-to-1”;
```

example Compressed Sound Header

```
myCompressedSoundHeader.samplePtr = nil;
myCompressedSoundHeader.numChannels = 1;
myCompressedSoundHeader.sampleRate = rate22khz;
myCompressedSoundHeader.encode = cmpSH;
myCompressedSoundHeader.numFrames = 7633
myCompressedSoundHeader.format = 0;
myCompressedSoundHeader.compressionID = threeToOne;
myCompressedSoundHeader.packetSize = threeToOnePacketSize;
myCompressedSoundHeader.snthID = 0;
myCompressedSoundHeader.sampleSize = 8;

// fill in the CompressionInfo from our Extended Common Chunk

 OSErr               err;
 CompressionInfo     cmpInfo;

 err = GetCompressionInfo(fixedCompression,
                          (OSType)(myExtendedCommonChunk.compressionType),
                          myExtendedCommonChunk.numChannels,
                          myExtendedCommonChunk.sampleSize,
                          &cmpInfo);

// fill in the CompressionInfo from our Compressed Sound Header

 OSErr               err;
 CompressionInfo     cmpInfo;

 err = GetCompressionInfo(myCmpSoundHeader->compressionID,
                          myCmpSoundHeader->format,
                          myCmpSoundHeader->numChannels,
                          myCmpSoundHeader->sampleSize,
                          &cmpInfo);
```

Note that this call will work for all sound formats, compressed or uncompressed. Using GetCompressionInfo() the Sound Manager will do the right thing. We get back the information we need in the CompressionInfo struct, with no special casing needed. Upon returning from the call to GetCompressionInfo() we have a filled CompressionInfo struct.

```
recordSize = 20
format = MAC3
compressionID = threeToOne
samplesPerPacket = 6
bytesPerPacket = 2
bytesPerFrame = 2
bytesPerSample = 1
```

We can now use this information to determine the duration of our sound.

```
duration = (numSampleFrames * samplesPerPacket) / sampleRate
```

By substituting the data given from the example above, we get the following results.

```
2.06 seconds = (7633 * 6) / 22254.54545
```

By including the CompressionInfo struct you should never need to special case code for compressed vs. uncompressed sounds -- and all sound data calculations should be correct. The following is a list of useful calculations which can be made using the data returned in the struct, along with data from our Extended Common Chunk or Compressed Sound Header:

```
seconds = (numFrames * samplesPerPacket) / sampleRate;
samples = numFrames * samplesPerPacket;
bytes = numFrames * bytesPerFrame;
compressionRatio = (samplesPerPacket * bytesPerSample) / bytesPerPacket;
```

GetCompressionName()

```
pascal OSErr GetCompressionName(OSType compressionType, Str255
compressionName)
```

GetCompressionName() returns a string describing the given compression format in a string that can be displayed to the user. The compressionType parameter specifies the compression format, and the name is returned in compressionName. This string can be used in pop-up menus and other user interface elements to allow the user to select a compression format.

SoundConverterOpen()

An architecture has been added to Sound Manager 3.2 to allow you to easily convert between sound formats. Some of the operations that can be performed are compression, decompression, channel conversion, sample rate conversion and sample format conversion.

A conversion session is begun by calling SoundConverterOpen(), to which you pass the format of the sound to be converted and the desired output format. A

SoundConverter identifier is returned that must be passed to all further routines in this session. SoundConverterClose() is used to close the session.

SoundConverterGetBufferSizes() allows you to determine input and output buffer sizes based on a target buffer size. This lets you allocate buffers to fit the conversion established with SoundConverterOpen().

Converting a sound is a three-step process. First, you call SoundConverterBeginConversion() to initiate the conversion and reset the SoundConverter to default settings. Then SoundConverterConvertBuffer() is called one or more times to convert sequential buffers of the input data to the output format. Finally, when all input data has been converted, SoundConverterEndConversion() flushes out any data left in the converter.

```
pascal OSErr SoundConverterOpen(const SoundComponentData *inputFormat,
             const SoundComponentData *outputFormat, SoundConverter *sc)
```

SoundConverterOpen() sets up the conversion session and returns a SoundConverter identifier to be passed to all further routines. The inputFormat parameter specifies the format of the sound data to be converted using a SoundComponentData structure. The following fields must be set up to describe the sound correctly:

```
flags           - set to 0
format          - the sound format (i.e. 'raw ', 'twos', 'MAC3', etc.)
numChannels     - the number of channels  (i.e. 1 = mono, 2 = stereo)
sampleSize      - the sample size (i.e. 8 = 8-bit, 16 = 16-bit)
sampleRate      - the sampling rate (in samples/second)
sampleCount     - set to 0
buffer          - set to 0
reserved        - set to 0
```

The outputFormat parameter specifies the output format, and must be passed fields similar to inputFormat. Output fields that are different from input fields will cause a conversion. For example, if the input sound format is 'raw ' and the output format is 'MAC3', the data resulting from the conversion will be compressed with MACE 3:1. This allows any combination of compression, decompression, channel conversion, sample size conversion and sampling rate conversion. A SoundConverter identifier is returned to manage the session, which must be passed to all further routines.

SoundConverterClose()

```
pascal OSErr SoundConverterClose(SoundConverter sc)
```

SoundConverterClose() terminates the session and frees up all memory and services associated with this session.

SoundConverterGetBufferSizes()

```
pascal OSErr SoundConverterGetBufferSizes(SoundConverter sc,
              unsigned long targetBytes, unsigned long *inputFrames,
              unsigned long *inputBytes, unsigned long *outputBytes)
```

SoundConverterGetBufferSizes() is used to determine the input and output buffer sizes for a given target size. This is so you can make sure your buffers will fit the conversion parameters established with SoundConverterOpen().

The targetBytes parameter is the approximate number of bytes you would like both your input and output buffers to be. The inputFrames and inputBytes parameters return the actual size you should make your input buffer, in frames and bytes respectively. The outputBytes parameter returns the size in bytes for your output buffer.

Note: The returned input and output buffer sizes **can** be larger than your target size settings. This is because they are rounded up depending on the format, but they will be very close to the target settings. Also note that the input and output sizes may be very different, depending on the input and output formats given in SoundConverterOpen. The sizes are calculated assuming you will convert all data in the input buffer to the output buffer.

SoundConverterBeginConversion()

```
pascal OSErr SoundConverterBeginConversion(SoundConverter sc)
```

SoundConverterBeginConversion() starts a conversion. All state information is reset to default values in preparation for a new input buffer.

This routine can be called at interrupt time.

SoundConverterConvertBuffer()

```
pascal OSErr SoundConverterConvertBuffer(SoundConverter sc,
                     const void *inputPtr, unsigned long inputFrames,
                     void *outputPtr, unsigned long *outputFrames,
                     unsigned long *outputBytes)
```

SoundConverterConvertBuffer() converts a buffer of data from the input format to the output format. The inputPtr parameter points to the input data, and inputFrames gives the number of frames in that buffer. The outputPtr parameter specifies where the output data should be placed. The output Frames and outputBytes parameters return the number of frames and bytes placed in the output buffer respectively.

This routine will consume all the data in the input buffer, but, depending on the complexity of the conversion, not all the converted data may be put in the output buffer right away. The SoundConverterEndConversion() routine is used to flush out all this remaining data before a conversion session is closed.

If you are using this routine in conjunction with SoundConverterGetBufferSizes(), it is very important that you do not pass in a value in inputFrames larger than the frames value returned by SoundConverterGetBufferSizes(), or you will overflow your output buffer. The SoundConverterConvertBuffer() calls converts ALL the input data!

This routine can be called at interrupt time.

SoundConverterEndConversion()

```
pascal OSErr SoundConverterEndConversion(SoundConverter sc, void *outputPtr,
                    unsigned long *outputFrames, unsigned long *outputBytes)
```

SoundConverterEndConversion() ends a conversion. Any data remaining in the converters is flushed out and returned here.

This routine can be called at interrupt time.

Conversion Example

The following is an example of how to use the sound conversion architecture to convert a buffer of silence to IMA 4:1, changing the sampling rate in the process.

```
enum {
    kTargetBytes = 20 * 1024
};

void main(void)
{
    SoundConverter      sc;
    SoundComponentData  inputFormat, outputFormat;
    unsigned long       inputFrames, inputBytes;
    unsigned long       outputFrames, outputBytes;
    Ptr                 inputPtr, outputPtr;
    OSErr               err;

    inputFormat.flags = 0;
    inputFormat.format = kOffsetBinary;
    inputFormat.numChannels = 1;
    inputFormat.sampleSize = 8;
    inputFormat.sampleRate = rate22050hz;
    inputFormat.sampleCount = 0;
    inputFormat.buffer = nil;
    inputFormat.reserved = 0;

    outputFormat.flags = 0;
    outputFormat.format = kIMA4SubType;
    outputFormat.numChannels = 1;
    outputFormat.sampleSize = 16;
    outputFormat.sampleRate = rate44100hz;
    outputFormat.sampleCount = 0;
    outputFormat.buffer = nil;
    outputFormat.reserved = 0;
```

```
    err = SoundConverterOpen(&inputFormat, &outputFormat, &sc);
    if (err != noErr)
       DebugStr("\pOpen failed");

    err = SoundConverterGetBufferSizes(sc, kTargetBytes,
                    &inputFrames, &inputBytes, &outputBytes);
    if (err != noErr)
       DebugStr("\pGetBufferSizes failed");

    inputPtr = NewPtrClear(inputBytes);
    outputPtr = NewPtrClear(outputBytes);

    // fill input buffer with 8-bit silence
    {
        int     i;
        Ptr     dp = inputPtr;

        for (i = 0; i < inputBytes; i++)
            *dp++ = 0x80;
    }

    err = SoundConverterBeginConversion(sc);
    if (err != noErr)
       DebugStr("\pBegin Conversion failed");

    err = SoundConverterConvertBuffer(sc, inputPtr, inputFrames,
                    outputPtr, &outputFrames, &outputBytes);
    if (err != noErr)
       DebugStr("\pConversion failed");

    err = SoundConverterEndConversion(sc,
                    outputPtr,&outputFrames, &outputBytes);
    if (err != noErr)
       DebugStr("\pEnd Conversion failed");

    err = SoundConverterClose(sc);
    if (err != noErr)
       DebugStr("\pClose failed");
}
```

Using the “SoundLib” shared library

If you are developing a PowerPC-native application and wish to call some of the new routines in Sound Manager 3.0 and later, you will need to link with the SoundLib shared library. This is because not all of the Sound Manager routine definitions are included in the InterfaceLib library currently built into Power Macintosh systems. The Sound Manager 3.1 extension installs a shared library with these missing routine definitions, and you use SoundLib to reference this library.

SoundLib is a “dummy” library, and just contains symbol references to the real shared library in the Sound Manager 3.1 extension. The SoundLib file should be used only for linking your PowerPC application – it should NOT be installed in the System Folder and is not to be given to users, as this can cause library conflicts. The SoundLib file simply provides the symbols that are then resolved from the Sound Manager 3.1 extension at run time. You should “weak” link with SoundLib and check in your application for the presence of Sound Manager 3.1 before calling one

of these new routines (see sample code above). SoundLib is a .pef file, not a .xcoff file.

Some of the routines defined in SoundLib include GetCompressionInfo(), GetSoundPreference(), SetSoundPreference(), UnsignedFixedMulDiv(), SndGetInfo(), SndSetInfo() and all the sound component interfaces needed when developing a native sound component.

Bug fixes

The following is a brief summary of bugs that have been fixed with various releases since Sound Manager 3.0 and is not a complete list. The intention here is to point out major areas of improvement that might affect a large number of applications.

Sound Manager 3.1

• ampCmd values are always scaled to volumeCmd value. This allows sound channels that are issued the ampCmd or sound resources with embedded amdCmd's (e.g. Simple Beep) to work properly.

• Reset alert sound channels to full volume since Simple Beep leaves it set to zero.

• Changed get siHardwareMute in PowerMacs to report if speaker is muted, and if headphones are inserted and muted. Always set the speaker and headphone volume according to preferences when registering. This caused SysBeep() to flash the menu bar if the speakers were muted and headphones were in use.

• A sound component within the Sound Manager extension calling Get/SetSoundPreference() from within it's register method failed.

• Unlock the preference file name string resource after loading it the first time.

• Sound components code resource is not marked locked, which would cause it to load low in the system heap. This reduces system heap compacting and purging during SndNewChannel().

• Do not call UniqueID() in SetSoundPreference() because there may not be any QuickDraw globals. The default ID will be 0 instead of the result from UniqueID().

• SndNewChannel() works better in low memory conditions. It may open a channel with more allocated in the application's heap, when in the past it would only allocate from the system heap. This was found to be a problem for many multimedia applications that have very large SIZE resource settings (which caused the system heap to be crushed).

• SndChannelStatus() was not setting the scCurrentTime correctly for compressed data. It now calls GetCompressionInfo() for the proper values.

• SndPlay() will play any type of sound resource on any type of sound channel. Previously “Simple Beep” would not play using SndPlay() with a channel that was allocated for the sampledSynth.

• Getting a sound preference using GetSoundPreference() could sometimes cause CloseResFile() be to called on a random value. GetResource() may not return the resource and not set ResErr. This behavior is documented in *Inside Macintosh.*

• SndRecord() would leak a very large handle if any errors occurred.

• Added support in SetupSndHeader() for arbitrary compression formats instead of just MACE 3 and MACE 6.

• In SPBCloseDevice() we call SBPStopRecording() if the device is busy. Otherwise the system will hang and forces the user to reboot the machine.

• When recording to disk, if you re-use one of the data buffer handles, it must be resized in case the sample rate or size changed. This fixes a heap corruption problem when doing multiple recordings to disk with different sample rates and/or sizes.

• When recording and playing from disk, removed the code that calls HPurge() and HLock() at interrupt time. This can confuse the Memory Manager to no end when you interrupt MoveHHi() so we only do this at non-interrupt time now. This could mean that handles are left around locked and non-purgable when recording is over, but there is not much we can do about this until someone calls SPBStopRecording() or SndStopFilePlay().

• Fixed a bug in SndStartFilePlay() with ‘snd ‘ resources where it was re-using a parameter block to do a close before that parameter block was finished with the read. Fixed by using a third parameter block for the close.

• SndPlayDoubleBuffer() was computing the wrong amount of silence to play if the sound was compressed. This has been fixed.

• Fixed rounding problems in the sample rate converter. It was overshooting the buffer and caused clicking when downsampling with integer step factors.

• Apple Sound Chip sound component supports software muting. You see this in the Volumes panel of the Sound control panel.

• Allow for the use of any sample rate when recording with MACE compression.

• When turning on the play-through feature on the PowerMacs to hear an audio CD, the speaker would mute and un-mute incorrectly.

• Work around input hardware problem where the input hardware would reverse the left and right stereo signals on Power Macs.

Sound Manager 3.2

• New version of InstallMoveHHiPatch() that will install on PowerMacs when the old Memory Manager is running. On PowerMacs with our new native PowerPC sound components we use more than 3k of the stack. The MoveHHi() in the new ROMs will only preserve 3k so we have to patch it to preserve more stack space.

• Use a nil sound output component reference to specify the default device when calling Get/SetSoundOutputInfo().

• Deal with left over samples during compression better by preserving them across calls to PlaySourceBuffer(). This allows sequential calls to a compressor with non-packet multiple buffer sizes to seam together without clicks.

• Fixed SetSource() in the format converter so it does not ask the source for 8-bit twos-complement data, which no other sound component supports. This fixes problems when the format converter is installed after the mixer and is asked to output 8-bit twos.

• In the sample rate converter the samplesToSkip field now stores the amount to skip into a new buffer. The byteOffset field is updated to this value when a new buffer is received. This fixes a problem playing scales with the freqDuration command when a new buffer is played before the old buffer has reached the end.

• The gestaltSoundAttr proc being installed needs to avoid using unknown CPUs or re-cycled machine types.

• Multiply bytesPerSample by 8 to get sampleSize for sound header. Fixes bug in SetupSndHeader with arbitrary compression formats.

• Fixed the siHardwareBusy selector in the 8500 series.

• Added speaker and headphone muting into the preferences. This allows for the user to specify the speaker is no muted when using external speakers or headphones, and the next time the machine is started the user's preference is restore. Previously it would default to always muting the speaker if something was inserted into the headphone jack.

• Preserve register A0 within sound input driver when calling getting preferences.

• SndSoundManagerVersion(), SPBVersion(), and MACEVersion() should all return a NumVersion. A recent version of the interfaces were changed to return an unsigned long, but this fails with creating native PowerPC code.

Sound Manager 3.2.1

• The sample rate converter saves the interpolation tap values across buffers. It was clearing this value when a new buffer was issued (e.g. bufferCmd) which caused a click between buffers.

• Fix siHardwareBusy selector for Apple Sound Chip and some PowerMac machines. It was returning a random result.

• Updated the type and creator of the SoundLib to the new constants. This avoids a problem with the Code Fragment Manager which cannot distinguish between a run-time shared library and a dummy link library. Note that this file is only for programmers to use when building PowerPC applications. This file does not belong within a System Folder.