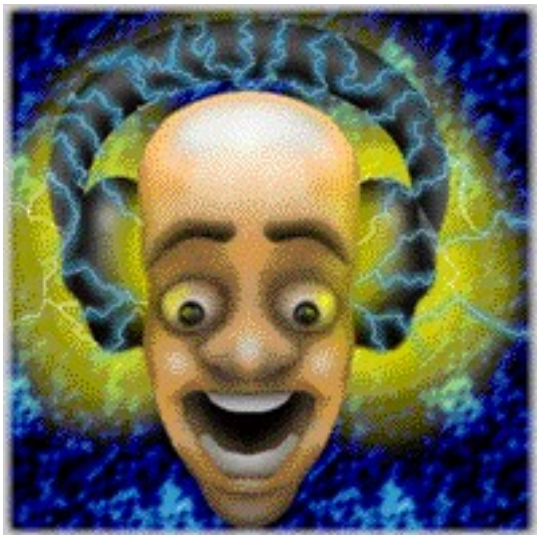

Business and Technical Contact

Steve Hales
14 Sunnyside Avenue
San Anselmo, CA 94960
office 415.258.9223
fax 415.258.9353
email hales@netcom.com

Last revision 2/8/94

SoundMusicSys
Technical Documentation

This document has essentially two parts: an **OVERVIEW** section, which provides a high-level description of the workings of the SoundMusicSys, and a **FUNCTION DESCRIPTIONS** section.

OVERVIEW

The following section describes the driver for basic, no-frills use. More complete descriptions of parameters and functions are located under **FUNCTION DESCRIPTIONS.**

THE BASICS

Here's a step-by-step method for playing music and sound.

Which libraries to link depends upon the compiler and options that you are using. Here's a basic guide.

In all cases you should include "SoundMusicSystem.h" whenever you want to use any functions or features of SoundMusicSys.

For Symantec's 68k Think C compilers. Include into your project
'SoundMusicSystem (Rev 3).lib' for A5 based projects.
'SoundMusicSystem (Rev 3).lib A4' for A4 based projects.

For Metrowerks CodeWarrior compilers. Include into your project:
For 68k projects:
'MWSoundMusicSystem(rev3).lib' for A5 based projects.
'MWSoundMusicSystem(rev3).lib A4' for A4 based projects.
For PowerPC projects:
'MWPPCSoundMusicSystem(rev3).lib'.

For MPW compliers. Link into your projects the following library:
For 68k projects:
'MPWSoundMusicSystem(rev3).lib'.
For PowerPC projects:
'MPWSoundMusicSystem(rev3).xcoff'.

In your resource fork of your project, you must include the 'MDRV' resource,

and ‘SMOD’ resources.

A note about Application development verses INIT / XCMD or Desk Accessory development. Since Applications reference globals via the A5 register and INIT’s and Desk Accessories use A4, you must include the “SoundMusicSystem (Rev 3).lib A4” for THINK C. ‘MWSoundMusicSystem(rev3).lib A4’ for Metrowerks. MPW does not allow A4 development, all globals are referenced by A5.

• First perform a one-time initialization of the driver. (In order to determine appropriate values for initialization, you can call **MaxVoiceLoad()** first).

```
theErr = InitSoundMusicSystem(    5,      // Max song voices
                                  5,      // Max normalized voices
```

2, // Max sound effects track
jxLowQuality);

Once this call is made, you can play music, but not sound effects.

PLAYING SOUND EFFECTS

You must register sound effects with the driver prior to playing them. Sound effects are referenced by resource ID number, and are stored as type 1 or type 2 'snd' resources, or as ‘csnd's,’ SoundMusicSys's proprietary compression format.

• To register sound effects, call:

static short int theSounds[] = {127, 128, 129, 130, SOUND_END};

RegisterSounds(theSounds, FALSE);

This loads the sounds with 'snd' resource numbers 127, 128, 129, and 130 into memory for play on demand. The FALSE passed to RegisterSounds forces the sounds to be loaded immediately. TRUE delays the loading of the sounds until BeginSound() is called. (q.v. Execution Control under Sound Effect Functions).

• To play a sound effect, call:

BeginSound(128, 11127 << 16L);

This plays ‘snd’ resource 128 at a rate of 11127 Hz. (and loads it if not already loaded). (q.v. Execution Control under Sound Effect Functions).

• To see if any sound effects are playing, call:

IsSoundFXFinished();

Note that no parameter is passed. (q.v. Execution Status under Sound Effect Functions).

• To terminate a sound effect, call:

EndSound(128);

This call interrupts the sound effect with 'snd' resource ID number 128. It remains resident in memory, and must be freed by the programmer. (q.v. Memory Management under Sound Effect Functions).

• To release the memory used by current set of sound effects, call:

ReleaseRegisteredSounds();

At this point any requests to play sound effects (using BeginSound()) will be ignored by the driver until you call RegisterSounds with a new set of sound effects.

• To shut down the driver, call

FinisSoundMusicSystem();

Do this just prior to exiting your application. (By the way, this is the correct spelling of the function's name).

PLAYING MUSIC

Music is easy to manage, provided that you have correctly set up a 'SONG' resource containing the necessary 'INST', 'Midi', and snd resources. There is currently no means to load multiple songs for playback on demand.

• To play a song (assuming you have already initialized the system with **InitSoundMusicSystem()**), call

BeginSong(128)

This plays the song with song resource ID 128.

• To stop the current song, call

EndSong()

Any resources associated with the song are deallocated by this call. There is currently no way to stop a song without deallocating its resources.

ADVANCED USE

MULTITASKING

You can pause the SoundMusicSystem so that your application and others have access to the Apple Sound Manager. (e.g. to play the system beep).

• To suspend the SoundMusicSys, call

PauseMusicSystem().

• To continue, call

ResumeSoundMusicSystem();

SOUND LISTS

There is a mechanism for creating lists of samples to be played asynchronously in a predetermined order . Here is a data list defining 5 sounds and their playing time in 60ths of a second:

```
static SampleList playList[] = {    {127, SOUND_RATE_DEFAULT, 130},
                                    {128, SOUND_RATE_DEFAULT, 130},
                                    {129, SOUND_RATE_DEFAULT, 120},
                                    {130, SOUND_RATE_DEFAULT, 30},
                                    {1000, SOUND_RATE_DEFAULT, 150}
```

};

• You can then play the sounds with:

BeginSoundList(&playList, 5);

• To end **all** of these sounds, call (with no parameter passed):

EndSound();

All sounds remains resident in memory, but are purgeable. (q.v. SoundLock() under Memory Management in the Sound Effect Functions section).

• To see if the sound list is playing call:

IsSoundListFinished();

You can use the callback system (q.v. Execution Status under Sound Effect functions) to tell you when a specific sound effect is finished playing, or query the system about a specific sound effect. Currently there is no way to ask the driver what sound is currently playing.

COMPRESSION

The application "CompressSnd" is based upon the public-domain version of the LZSS compression algorithm,The CompressSnd utility will ask for a source file and a destination file; it examines all 'snd' resources found in the source file and tries to compress them. The application will create a 'csnd' resource for each corresponding 'snd' resource that is successfully compressed; when unsuccessul, it will copy the original 'snd' into the destination file. A 'csnd' is decompressed when loaded into memory. The rate of decompression is about 800k per second on a 16 MHz 68020, with no loss in quality. If the SoundMusicSys is told to purge the sounds, they will be made available for purging after they have been played.

MACE 3:1 and 6:1 compression of 'snd' resources is now supported.

PLAYING UNREGISTERED SOUND EFFECTS

The driver expects things to be in snd format; let's say you have your sounds in a format other than 'snd.' You can play samples without having to register

them with the driver by using the function PlayTheSample. You pass a buffer, the length of the buffer and the playback rate. The buffer must have an extra 6 bytes at the beginning for internal use by the driver.

Here is a call with a 1000 byte buffer to be played at 22732 Hz:

PlayTheSample(pBlock, 1006L, 22732 << 16L);

The drawback of this method is that you do not have access to all of the cool driver functions.

VIRTUAL MEMORY

The SoundMusicSys Sound Driver will work with Virtual Memory. However, the Virtual Memory Manager could swap out a page containing data the driver needs; this will cause pauses in playback. All efforts have been put into place to minimize the impact of a page swap, but until Apple allows some interrupts to take priority the Virtual Memory manager will always stomp on the SoundMusicSys driver. It is not advisable to use Virtual Memory.

POWERPC

The SoundMusicSys driver works as a accelerated resource. Glue code that is linked into your projects are native, and the MDRV resource currently is 68k code. As the full native version of the MDRV resource becomes available, you will be able to swap out the old MDRV for the new native one. At the moment a complete native version is hampered by the fact the Apple's Sound Manager is emulated 68k code.

FUNCTION DESCRIPTIONS

WHAT THE DRIVER CAN'T DO

Currently the driver cannot perform the following functions:

- Can't end a song and keep the associated resources in memory. All resources are deallocated when a song ends.
- Can't pre-load multiple songs; you can only have one song in memory at a time.
- Can't ask the driver what sfx it is currently playing.
- Can't determine how much memory is required to load a song.

SYSTEM-LEVEL CONTROL

OSErr InitSoundMusicSystem(short int maxSongVoices,
short int maxNormalizedVoices,
short int maxEffectVoices,
SoundQuality quality);

Call this ONCE at the beginning of your application. If you don't plan to play any music, then pass 0 for maxSongVoices.

maxSongVoices

The maximum number of voices available for playing music. When this limit is exceeded, the oldest notes will be eliminated first. For playing only sound effects and no music, set maxSongVoices to 0.

maxEffectVoices

The number of voices available for sound effects. When sound effects overlap, a new sound effect will interrupt an old one.

maxNormalizedVoices

This number represents an inverse gain level: the driver doles out the dynamic range among the indicated number of Normalized voices, so an increase in the number of normalized voices yields a decrease in the dynamic range of each voice. Set maxNormalizedVoices to the number of voices needed to play most of the music. For instance, your music uses 8 voices maximum, 4 voices most the time, plus one sound effect at a time: a value of 5 for maxNormalizedVoices is appropriate. When additional voices are needed, then the overall volume will decrease because the dynamic range is being doled out to a larger number of voices. If you set

maxNormalizedVoices too low, then there will be drastic changes in volume and signal to noise ratio. (You can prevent drastic fluctuations by telling the driver in the song resource to drop notes if maxNormNotes is exceeded).

SoundQuality is one of five values:

jxAnalyzeQuality - Let the driver determine the best driver to use.
jxLowQuality - Lowest quality (11khz)
jxHighQuality - Best quality (22khz)
jxInterpLowQuality - Lowest quality with 2 pt interpolation
jxInterpHighQuality - Best quality with 2 pt interpolation
jxInterpBestLowQuality - uses more memory and CPU time
jxInterpBestHighQuality - uses the most memory and CPU time

You can let the driver determine the best quality by passing **jxAnalyzeQuality**. The 22 kHz output takes approximately twice the CPU time.

NOTE:

The interpolation buffer for **jxInterpBestLowQuality** and **jxInterpBestHighQuality** (between 32 and 128K) is not deallocated until you shut down the system with FinisSoundMusicSystem. Maintaining this buffer prevents heap fragmentation.

Activation of the SoundMusicSys sound driver deactivates the Apple Sound Manager, unless you are running Sound Manager 3.0 or better. Pre 6.0.7 versions of the OS only allow one sound channel open at once. If you want to have that channel free, but have the sound driver initialized, then call PauseSoundMusicSystem right after calling InitSoundMusicSystem. When you want to play a song, call ResumeSoundMusicSystem. (Note: this step is not necessary for Sound Manager 3.0 or better).

Check for errors by comparing against “noErr” or zero. Errors can be

positive or negative numbers.

The following errors that can be retuned from InitSoundMusicSystem or ChangeSystemVoices and thier possible meanings.

noErr No errors.
memFullErr Out of memory.
resProblem Resource problem. Check that the 'MDRV' resource is available.
dsOldSystem old system need at least System 6.0 or greater.
paramErr Parameter error. SoundQuality was invalid, maxSongVoices was less than maxNormalizedVoices.

OSErr ChangeSystemVoices(short int maxSongVoices,
short int maxNormalizedVoices,
short int maxEffectVoices)

This function allows you to change the current voice setup of the driver. The function BeginSong() calls this function.

Errors returned can be the same as InitSoundMusicSystem.

maxSongVoices

The maximum number of voices required for a particular song. When this limit is exceeded, notes will be dropped on a FIFO basis, i.e. the oldest note will be eliminated first.

maxEffectVoices

The maximum number of voices available for creating sound effects.

maxNormalizedVoices

The normalized number of voices that are used at once. i.e.. if your music is uses 8 voices Max, but only 4 most of the time, and you want only one sound effect at a time, then set maxNormalizedVoices to 5. When the number of voices exceeds maxNormalizedVoices then the overall volume will decrease (unless you tell the driver in the song resource to drop notes when maxNormNotes is exceeded).

If you call ChangeSystemVoices while the system has been paused with PauseSoundMusicSystem the error channelBusy will be returned.

NOTE: This will stop all sounds that are playing; songs will be deallocated, but not sound effects.

void FinisSoundMusicSystem(void);

Call this ONCE at the end of you application. This will clean up and deallocate any memory used.

OSErr PauseSoundMusicSystem(void);

Call this when you want to release the sound hardware for a moment. Good for handling suspend event from the event manager.

OSErr ResumeSoundMusicSystem(void);

Call this when you want to use the sound hardware again. Good for the resume event from the event manager. If you get a channelBusy error, then the Sound Manager is still busy. ALWAYS CHECK for errors with this function.

MUSIC FUNCTIONS

CONTROL

OSErr BeginSong(short int songID)

The **songID** identifies a 'SONG' resource, which contains information telling the driver which 'INST' and ‘snd/csnd’ resources to load. The song plays as soon as the buffer-ahead buffer is filled. You can pre-roll, or pre-load a song by calling LoadSong first, then call BeginSong to start the song. If you call LoadSong and BeginSong with a different ID, the new song will be loaded, and the old song's resources deallocated.

The following errors are returned:
resNotFound is returned if the ‘SONG’ resource cannot be found.

fidNotFound is returned if the ‘Midi/cmid’ resource cannot be found

protocolErr is returned if you call BeginSong during an interrupt.

A positive error maybe returned if there is a problem loading the instruments.

OSErr BeginSongLooped(short int songID)

Same as BeginSong, except the driver loops through the data in 'Midi/cmid' until EndSong() is called.

OSErr BeginSongFromMemory(short int songID, Handle theSong, Handle theMidi Boolean loopSong)

Same as BeginSong, except this allows you to pass a handle of a 'SONG' and 'Midi' resource. This is very useful for creating one 'SONG' resource for many 'Midi' resources. The 'Midi' resource is just a standard Midi file. Passing TRUE to loopSong will cause the song to continue to play until EndSong is called.

void EndSong(void)

This ends the current song, and frees up song related resources.

MUSIC MEMORY MANAGEMENT

Note: Currently, there is no function which tells the programmer how much memory is required to load a song.

void PurgeSongs(Boolean)

If set to TRUE, when a song is finished playing it will be purged from memory. FALSE allows the song resource to stay around.

void LockSongs(Boolean)

If set to TRUE, a song's resources will be Locked when the song finishes. FALSE leaves the resources unlocked.

void LoadSong(short theID)

Pre-loads a song for future use. Call BeginSong or related call with the same song ID to play the song. Otherwise, the song with the new ID will be loaded and played.

void FreeSong(void)

Stops and deallocates the last song that was played or pre-loaded (with LoadSong).

Handle DeltaDecompressHandle(Handle theCompressedData)

This function decompresses a ‘csnd’ resource. The returned handle is that of a ‘snd’ resource. You do not have to initialize the driver before using this function.

MUSIC EXECUTION STATUS

void Boolean IsSongDone(void)

Returns FALSE if a song is playing. There is no way to check if a song is loaded but not playing.

long SongTicks(void)

SongTicks returns the number of Ticks (1/60th of a second) since the beginning of the song.

MACHINE CAPABILITIES

short int MaxVoiceLoad(void)

Returns the number of voices the Sound Driver can handle based upon the Macintosh Hardware.

16 For 68040 Machines.
8 For 68030 Machines.
6 For 68020 Machines.
3 For 68000 Machines.

These numbers are estimates, and carry the assumption that the Sound Driver will not consume more than 50% of the CPU time. Using more than 50% of the CPU time can crash the machine.

Note: you can call this function before initializing the driver with InitSoundMusicSystem.

FADING & MIXING

short int GetMasterVolume (void)

This function reports the state of the Macintosh hardware, not the current driver volume. Call this function at the beginning of your application, and store the value. When pausing or exiting your application, call SetMasterVolume will this stored value; this ensures that original system volume remains unaffected.

void SetMasterVolume (short int theVolume)

This function sets the volume of the Macintosh hardware; two important volume levels are NO_VOLUME and FULL_VOLUME. Smooth volume changes are not guaranteed, however.

void BeginMasterFadeOut(long time)

This will take the current sound output, the combination of Sound effects and music, and fade it out in the specified number of 1/60th of a second (**time** * 1/60 seconds). For example: If you pass 60 to BeginMasterFadeOut, then all the sound will be silence in 1 second. After that time, the output level is brought back to normal. If you need to reset the volume before the fade is finished, call SetMasterFade(FULL_VOLUME).

void BeginMasterFadeIn(long time)

This will take the current sound output, which is a combination of Sound effects

and music, and fade them in by the given number of 1/60th of a second. For example: If you pass 60 to BeginMasterFadeIn, then all the sound will be at full volume in 1 second. If you need to set the volume before the fade is finished, call SetMasterFade(FULL_VOLUME).

long SetMasterFade(long fadeLevel)

Will set the master fade level. 0 is silence, 256 is full volume.

long FadeLevel(void)

FadeLevel returns the current fading level that BeginMasterFadeOut starts at. Maximum output is 256, silence is 0.

OSErr ChangeOuputQuality(SoundQuality)

ChangeOuputQuality allows you to change the output quality on the fly, rather than re-initialize with InitSoundMusicSystem. Pass either jxHighQuality or jxLowQuality if you don't want sound playback interrupted; passing a parameter that turns on interpolation will interrupt playback, since the driver has to allocate a buffer.

FILE PLAYBACK

OSErr StartFilePlayback(FSSpec *theFile, long theRate, long bufferSize)

Given an FSSpec (a file specification), this function streams sound data from a file.

• The bufferSize is temporary storage space to maintain smooth streaming. BufferSize must be at least 10,000 bytes (actual amount of memory allocated is twice bufferSize). The bigger the buffer, the better the probability that the data stream won't hiccup; the slower your storage device, the bigger your buffer should be.

• The file must be 8-bit unsigned data (no header, just waveform data).

• The sample rate for playback is theRate. This is the same parameter passed into BeginSound.

• StartFilePlayback will loop at the end of the file. You can use the callback mechanism to determine when the end of the file is reached.

• Errors returned vary from File Manager errors, to Memory manager errors.

Note: The sound IDs for the file streaming are defined as FILE_PLAY_1_ID and FILE_PLAY_2_ID.

When you call EndAllSounds, or StartFilePlayback again, the current file being streamed will be shut off.

void EndFilePlayback(void)

After calling StartFilePlayback, use this call to stop the playback and free the channel.

void ChangeFilePlaybackRate(long theNewRate)

This call provides the means to change on the fly the playback rate of the file

being streamed from disk.

Boolean ServiceFilePlayback(void)

This function checks to make sure that there is data to played; if not, it gets more from drive storage. Call this function in your main event loop, NOT during an interrupt. Not calling this function frequently enough can cause the file playback mechanism to stutter.

ServiceFilePlayback returns two possible values:
FALSE: nothing happened (didn't need to get more data)
TRUE: successfully read a new stream of data.

FALSE will be returned most of the time.

SOUND EFFECT FUNCTIONS

SOUND REGISTRATION

void RegisterSounds(short int *pSoundID, Boolean registerOnly);

Loads memory with a sound effect list if **registerOnly** is FALSE. Otherwise, the sounds are loaded when BeginSound is called. Every call to RegisterSounds must be matched by a corresponding call to ReleaseRegisteredSounds.

void ReleaseRegisteredSounds(void);

Call this to free memory when you are done with a particular set of sound effects. Used in conjunction with RegisterSounds, which allocates memory for a new set of sounds. You can dynamically allocate memory with these two functions, so that you don't have to have all your sound effects in memory or in one file.

SOUND EXECUTION CONTROL

OSErr BeginSound(short int theID, long theRate);

Plays snd resource **theID**. This loads the sample into memory if it has been purged. The rate is passed as a 16.16 Fixed point value denoting samples-per-second, e.g. the rate 11127.0 khz would be 729219072 (11127 << 16L). Use SOUND_RATE_DEFAULT if you don't know the correct playback rate:

BeginSound(128, SOUND_RATE_DEFAULT);

If there is no memory available to load the sound, BeginSound returns **memFullErr** (-108).

void EndSound(short int theID);

This stops the sound with resource ID **theID**, but it remains in memory. If you want to stop all sound effects, including sample lists, call EndAllSound().

void BeginSoundList(SampleList * sampleList,
short int totalSamples);

Sequentially plays a list of sounds. Because of the way these samples are played, BeginSoundList loads all the samples into memory before playing any of them. Make sure you have enough free memory available before starting a list. (q.v. **SoundMemorySize ()**)

void EndSoundList(void);

This ends the current sound list. It remains in memory.

void EndAllSound(void);

This ends all sound effects that are currently playing, including sound lists and song. Sound effects remain in memory, but song resources are deallocated.

void BeginSoundSection(short int theID, long theRate, long secStart, long secEnd);

Exactly like BeginSound, except that you specify the beginning and end samples. Useful for multiple packed sounds.

void BeginSoundLoop(short int theID, long theRate, long loopStart, long loopEnd);

Exactly like BeginSound, except that you specify the beginning and end samples of a loop. The sound loops until another sound is started or EndSound() is used.

void BeginSoundReverse(short int theID, long theRate);

Works the same as BeginSound, but will play the sample in reverse.

Macros

SOUND_RATE_DEFAULT

Sets the playback rate to that contained in the 'snd' resource. This is the most common way to use this system.

SOUND_RATE_FAST, SOUND_RATE_22k

Sets playback rate to 22,254 Hz (samples per second).

SOUND_RATE_MEDIUM, SOUND_RATE_11k

Sets playback rate to 11,1127 Hz.

SOUND_RATE_MOSEY, SOUND_RATE_7k

Sets playback rate to 7,418 Hz.

SOUND_RATE_SLOW, SOUND_RATE_5k

Sets playback rate to 5,563.5 Hz.

void PlayTheSample(Ptr pSamp, long sampSize, long sampRate);

This function allows you to create sounds to play through the sound system. Be sure to add an extra 6 bytes at the beginning of your buffer before you call PlayTheSample().

Note: The sound ID for the custom buffer play is defined as CUSTOM_PLAY_ID.

EXECUTION STATUS

Boolean IsSoundFXFinished(void);

If any sound is playing, this function returns FALSE.

Boolean IsThisSoundFXFinished(short theID);

Once a sound has been started with BeginSound(), this returns TRUE if the sound **theID** is finished playing.

Boolean IsSoundListFinished(void);

After calling BeginSoundList this function will return TRUE if that last is finished playing.

void SetSoundDoneCallBack(void * theProc);

This function allows you to attach a call back system to notify you when sound effects are finished playing.

```
void MyCallback(short soundID)
{
        if (soundID == myID)
        {
                /* Do something interesting */
        }
}
```

The system will call your function (**theProc**) when a sound or a sound buffer is finished. The soundID passed is the same ID passed into BeginSound. If you are using PlayTheSample() buffer system, then the ID will be the constant CUSTOM_PLAY_ID. Your function is called at interrupt time, so be sure not to move memory or call any Toolbox traps that might move memory. The A4/A5 globals will have been setup before your callback is called.

Never use BeginSound or a variant thereof inside a call back function. These functions move memory, and can corrupt the heap. You can use PlayTheSample safely, since the memory is already allocated.

void SetSoundVBCallBack(ProcPtr theProc);

Every 1/60th of a second your function will be called. The function is just a standard function with no parameters. Make sure that your function does its job quickly, otherwise things will come to a halt. Register A4/A5 will be set up correctly, but don't use Toolbox traps that move or allocate memory.

MIXING and MUTATING

void ChangeSoundPitch(short int theID, long theRate);

Immediately changes the playback rate of a particular sound effect to **theRate**.

void ChangeSoundVolume(short int theID, short int theVolume);

Immediately changes the playback volume of a particular sound effect to **theVolume**. The range is from NO_VOLUME to FULL_VOLUME.

void BeginSoundEnvelope(short int theID, long theRate,
short intloopCount);

This plays the sound from the beginning, but uses the looping variables stored in the 'snd' resource. The function plays the loop **loopCount** times, then completes the sound. If passed -1 for **loopCount**, the sound loops until stopped by EndSound() or EndAllSound().

void BeginSoundEnvelopeProc(short int theID, long theRate,
void *loopProc);

This will loop the sound until the function loopProc returns TRUE, then play to the end of the sound. This can be used asynchronously.

MEMORY MANAGEMENT

Sounds are not automatically purged; it is the responsibility of the programmer to purge all sounds. You must call FreeSound() or PurgeAllSounds() to deallocate memory.

OSErr LoadSound(short theID)

Pre-loads a sound but does not play it. TheID is the ID of the sample that has been registered with RegisterSounds. If there is no memory available to load the sound, LoadSound() returns **memFullErr** (-108).

void FreeSound(short theID)

Allows you to free the memory occupied by a specific sound, rather than the entire list. TheID is the sample ID that has been registered with RegisterSounds.

void SoundLock (short in theID, Boolean flag)

This function locks the specified sound in memory; it will not get purged until you call SoundLock with a FALSE.

TRUE: lock sound in memory.
FALSE: make sound available for purging.

You can only use this function with sound effects.

void PurgeAllSounds(unsigned long minMemory);

The sound system will purge sounds until free memory goes below the specified memory limit. Put this call in your main loop; the time penalty to check the available is minimal. Do not call this function during an interrupt because it can corrupt the heap. Call BeginSound() to load them for use.

long SoundMemorySize(short int *pSoundID);

This function returns how many bytes are required for a sound effects list to be loaded into memory, you can anticipate your memory needs. This assumes that you want the entire sound list loaded simultaneously. With **registerOnly** set to TRUE (cf. **RegisterSounds()**), you can initialize sounds without having them all be in memory .

Note that there is no corresponding function for determining the memory requirements of a song.

RESOURCE ATTRIBUTES

short int CalcPlaybackLength(long theRate, long theLength);

Returns the length of time (in 60ths of a second) needed to play a sample .

Byte * GetSoundWaveform(short int theID);

Given a snd resource ID that has been registered with the sound system via RegisterSounds, this will return a pointer to the beginning of the sample waveform data. If the sound is not in memory, this function will load it.

long GetSoundLength(short int theID);

Given a snd resource ID that has been registered with the sound system via RegisterSounds, this will return the length of the sample waveform. If the sound is not in memory, this function will load it.

short int GetSoundDefaultRate(short int theID);

Given a snd resource ID that has been registered with the sound system via RegisterSounds, this will return the default rate of the sample waveform. If the sound is not in memory, this function will load it.

short int GetSoundTime(short int theID, long theRate);

Given a registered 'snd' resource ID (via RegisterSounds) and a playback rate, this function returns the number of 1/60th of a second that it will take to play the sound. If the sound is not in memory, this function will load it.

long GetSoundLoopStart (short int theID)
long GetSoundLoopEnd (short int theID)

Use these functions when you want to play the loop section of a sample. Here's a piece sample code demonstrating their use:

```
BeginSoundSection (128, 11127<<16L, GetSoundStartLoop(SoundID),
                                    GetSoundEndLoop(SoundID));
```

snd/csnd Resource Types

'snd' resources are 8-bit digital samples used for instruments and sound effects. Type 1 and 2 'snd' formats are supported. MACE data compression is now supportted. 3 to 1 and 6 to 1.

'csnd' resources are compress 'snd' resources. The CompressSnd utility asks for a source file and a destination file; it examines all 'snd' resources in the source file and tries to compress them. CompressSnd creates a 'csnd' resource for each corresponding 'snd' resource that is successfully compressed; when unsuccessul, it will copy the original 'snd' into the destination file.

You can reduce the size of your sampled instruments by looping them. A loop is a section of a sample which is repeated when a note's duration exceeds the length of the sample. You must use a digital sample editor (like Alchemy or SoundEditPro) to set the loop points for a sample. Save your samples in AIFF format, then convert them to resource format with the utility 'AIFF to Resource'; they must be in resource form to stick them into your driver resource with your resource editor.

Sample Loops

Loops must be at least 370 bytes long (Macintosh restriction). Larger loops obviously make for larger samples, but require less driver overhead to play long notes.

WARNING: When you play a sample above its sampled pitch, the loop shrinks (!) If you play the sample so that the loop is effectively smaller than 370 bytes, you will get clicks and pops. In general, do not play samples above their sampled frequency, unless they have large loops.

Be sure to follow Apple's restrictions that 'snd' resource ID numbers should not be lower than 4100. (The driver will function even if you don't follow this guideline).

Index

WHAT THE DRIVER CAN'T DO 6