THE SOUND MANAGER

See Also Macintosh Technical Notes #168, and #208

Audio Interchange File Format specification

Macintosh Audio Compression/Expansion specification

Written By Jim Reekes October 2nd, 1988

SOUND ADVICE

This document describes the System 6.0.2 Sound Manager. The original chapter describing the Sound Manager is ambiguous, inaccurate, and often contradicts itself. This chapter hopefully will clear up the confusion and get developers using the Sound Manager as was originally intended. This document replaces the Sound Manager chapter originally published in *Inside Macintosh Volume V*.

The Sound Manager is a replacement for the older Sound Driver documented in Inside Macintosh Volume II. The abilities of the Sound Driver are currently supported by the Sound Manager and it will utilize future hardware improvements. The Sound Manager offers more flexible ways of doing things and, includes new features and options, all requiring less programming effort. Many applications do not require the use of sound and therefore do not need to be concerned with the Sound Manager. Refer to the *Human Interface Guideline: The Apple Desktop Interface* when using sound.

A fundamental knowledge of music and sound synthesis is presumed in this document. There are utilities available from third parties that aid in the development of creating sampled sound resources. Creating wave table data or discussing the abilities of wave synthesis versus sampled sound synthesis is not covered in this document. Two

good reference books are *Computer Music, Synthesis, Composition, and Performance* by Charles Dodge and Thomas A. Jerse, and *Principles of Digital Audio* by Ken Pohlman.

This document contains an overview of the Sound Manager, and a detailed description of sound resources, routines and commands. All of the known bugs and limitations are collected into one section, “The Current Sound Manager”. A bug icon is used to point out information contained in this section that is relative to the text being read. For example, when reading about a sound command if a bug icon is shown, make sure you have read the “Current Sound Manager” section regarding that command.

TABLE OF CONTENTS

INTRODUCTION

The Sound Manager is a collection of routines that can be used to create sounds without knowledge of, or dependence on, the hardware available. By using the Sound Manager, applications are assured of upward-compatibility with future hardware and software releases. The Sound Manager will always take advantage of hardware advancements. Applications using the Sound Manager now will gain those advantages. When a command is sent to the Sound Manager, it is really a request. For example, if sound code written to play on a Macintosh II is being used on a Macintosh Plus or Macintosh SE (which have slower CPU clocks and less capable audio hardware) the Sound Manager will use synthesizers fitted best to those machine's abilities. Conversely, future Macintoshes may have improved audio hardware, and that same code will be utilized by the Sound Manager to take full advantage of these as-yet-undetermined hardwares. All of this is transparent to the application, yet serves to make that application compatible with the full line of Macintosh computers, present and future.

A *synthesizer* is very similar to a device driver. A synthesizer is the code responsible for interpreting the most general sound commands and using the hardware available to produce it. A synthesizer is stored as a resource which the Sound Manager will install. Customized synthesizers are supplied for every Macintosh configuration. Only one synthesizer can be active at any time. Apple's sound hardware is only supported when used with Apple's synthesizers. Writing synthesizers for Apple's hardware is not supported. Writing custom synthesizers for non-Apple hardware is beyond the scope of this document. All references to synthesizers in this document pertain to the Apple synthesizers that are supplied with the Sound Manager.

Modifiers are used to perform pre-processing of commands before they are received by a synthesizer. Modifiers can ignore, alter, remove, or add commands, or perform periodic functions. A modifier is a procedure in memory, or a resource which the Sound Manager can install. For example, if the application wanted to play a melody transposed up by an octave a modifier could be used to replace notes

with notes that are an octave higher.

Instructions for a synthesizer and modifier are sent through a command queue called a *sound channel*. Sound channels provide a means of linking applications to the audio hardware. The application provides a sequence of commands which are processed through a number of modifiers (if any) and finally through a synthesizer that creates the sound with the hardware.

USING THE SOUND MANAGER

The Sound Manager code that runs on the Macintosh Plus is the same that is used on the Macintosh SE. The code running on the Macintosh II is different, since it has the Apple Sound Chip installed. The Apple Sound Chip was developed to reduce the CPU's involvement with producing sound and to extend the capabilities of the Sound Manager.

The Sound Manager requires the use of the VIA1 timer T1. This conflicts with some third party MIDI drivers. As such, it is not possible to use both the Sound Manager and these MIDI applications.

There are two types of resources used by the Sound Manager, `'snd '` and `'snth'`. A `'snd '` resource contains data and/or commands. A `'snth'` resource is code used as a synthesizer or modifier to interpret the commands sent into a channel. Generally, applications only need to be concerned with `'snd '` resources. More information on the formats of `'snd '` resources and their use is given later.

The Sound Manager provides a range of methods for creating sound on the Macintosh. Most applications will only need to use a few of the Sound Manager routines. At the simplest end of the range is the use of the note synthesizer to play a simple melody or `_SndPlay`. `_SndPlay` only requires a proper `'snd '` resource. Such a resource will contain the necessary information to create a channel linked to the required synthesizer and the commands to be sent into that channel. An application can use the following code to create a sound with this method:

```
myChan := NIL;
sndHandle := GetNamedResource ('snd ', 'myBeep');
myErr := SndPlay (myChan, sndHandle, FALSE);
```

For more complete control of the sound channel, an application can open a sound channel with `_SndNewChannel`. The application will then send commands to that channel with `_SndDoCommand` or `_SndDoImmediate`. When the application's sound is completed, the application closes the channel with `_SndDisposeChannel`.

The System Beep

The trap `_SysBeep` is a call to the Sound Manager. The sound of the System Beep is selected by the user in the Control Panel using the Sound `'cdev'`. Except for the “Simple Beep”, `_SysBeep` will be performed by the Sound Manager. If this sound is selected on a Macintosh that doesn’t have the Apple Sound Chip (i.e. the Macintosh Plus and SE), the beep will be generated by the original ROM code. This has the benefit of bypassing the Sound Manager and the potential conflict of third party MIDI drivers which both use the VIA1 timer T1. Thus, this conflict over the timer can be avoided by setting the System beep to the “Simple Beep” using the Sound `'cdev'` in the Control Panel.

If an application has an active synthesizer, then `_SysBeep` may not generate any sound. This is because only one synthesizer can be active at any time. On a Macintosh without the Apple Sound Chip (i.e. the Plus and SE) when the “Simple Beep” is selected the beep will be heard, since it bypasses the Sound Manager. Applications should dispose of their channels as soon as they have completed making sound, allowing the `_SysBeep` to be heard.

`_SysBeep` cannot be called at interrupt time since the Sound Manager will attempt to allocate memory and load a resource.

Refer to the section “Current Sound Manager” regarding `_SysBeep` on a Macintosh Plus and SE.

The Note Synthesizer

The note synthesizer is the simplest of all the synthesizers supplied with the Sound Manager. The sound produced by this synthesizer is based upon a square wave. An application cannot play back a wave form description or recorded sound when using this synthesizer. Very little set up is required to use this synthesizer. It also has the advantage of using little CPU time. It can be used for creating simple monophonic melodies.

The Wave Table Synthesizer

The wave table synthesizer will produce sounds based on a description of a single wave cycle. This cycle is called a wave table and is represented as an array of bytes describing the timbre (tone) of a sound. Applications may use any number of bytes to represent the wave, but 512 is the recommended

length since *the Sound Manager will re-sample it to this length*. A wave table can be pulled in from a resource or computed by the application at run time. To install a wave table in a channel, use the `waveTableCmd`. Up to four wave table channels can be opened at once allowing an application to play chords, melodies with harmonies and polyphonic melodies.

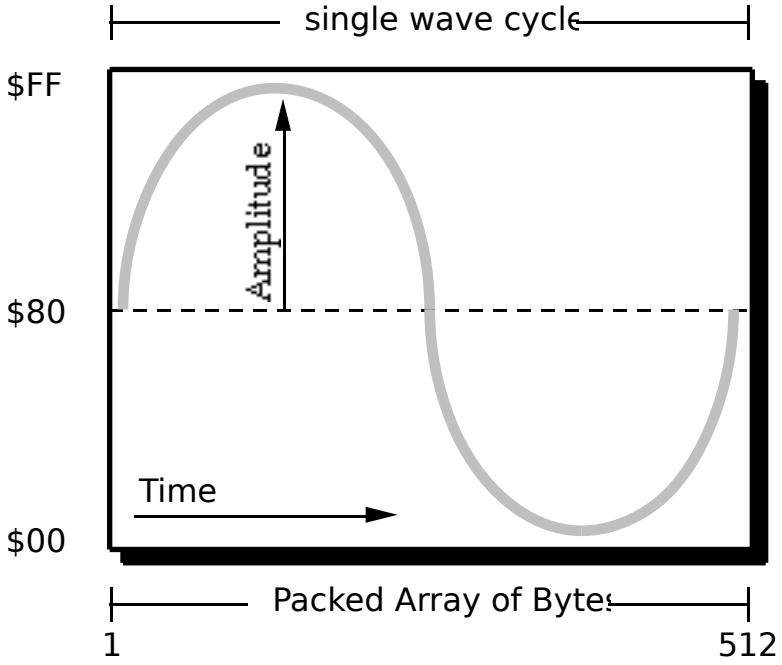


Figure 1 Graph of a Wave Table

A wave table is a sequence of wave amplitudes measured at fixed intervals. Figure 1 represents a sine wave being converted into a wave table by taking the value of the wave's amplitude at every 1/512th interval. A wave table is represented as a `PACKED ARRAY [1..512] OF BYTE`. Each byte may contain the value of `$00` through `$FF` inclusive. These bytes are considered offset values where `$80` represents a zero level of amplitude, `$00` is the largest negative value, and `$FF` is the largest positive value. The wave table synthesizer loops through the wave table for the duration of the sound.

Refer to the section "Current Sound Manager" regarding the wave table synthesizer on the Macintosh Plus and SE.

The Sampled Sound Synthesizer

The sampled sound synthesizer will play back digitally recorded (or computed) sounds. These sampled sounds are passed to the synthesizer in the form of a sampled sound header. This header can be played at the original sample rate, or at other rates to change its pitch. The sampled sound can be installed into a channel and then used as an instrument to play a sequence of notes. Thus a sampled sound, such as a harpsichord, can be used to play a melody. This synthesizer is typically used with pre-recorded sounds such as speech, songs or special effects. Developers concerned with saving sampled sound files need to refer to the Audio Interchange File Format available from the Apple Programmer's and Developer's Association. Figure 2 shows the structure of the sampled sound header used by the sampled sound synthesizer.

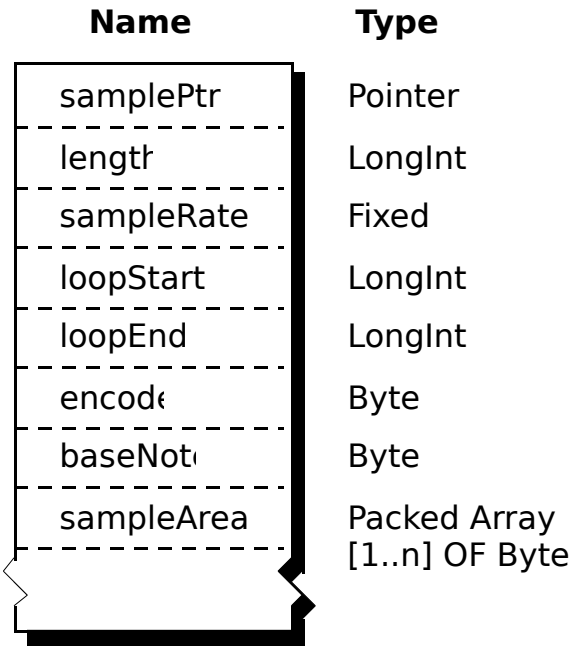


Figure 2 Sampled Sound Header

The first field of a sampled sound header is a `POINTER`. If the sampled sound is located immediately in memory after the `baseNote`, this field is `NIL`, otherwise it will be a pointer to the sample sound data. The `length` field is the number of bytes in the `PACKED ARRAY [1..n] OF BYTE` containing the sampled sound, `n` being this `length`.

RATE	DECIMAL	HEX
5kHz	5563.6363	$15BB.A2E8
7kHz	7418.1818	$1CFA.2E8B
11kHz	11127.2727	$2B77.45D1
22kHz	22254.5454	$56EE.8BA3
44kHz	44100.0000	$AC44.0000

Table 1 Sample Rates

The `sampleRate` is the rate at which the sample was originally recorded. These unsigned numbers are of type `FIXED`. The approximate sample rates are shown in Table 1.

The loop points contained within the sample header specifies the portion of the sample to be used by the Sound Manager when determining the `duration` of a `noteCmd`. These loop points specify the byte numbers in the sampled data used as the beginning and ending points to cycle through while playing the sound.

Refer to the section “Current Sound Manager” regarding the `noteCmd` and looping with a sampled sound header.

The `encode` option is used to determine the method of encoding used in the sample. The current `encode` options are shown below.

```
stdSH = $00 {standard sound header}
extSH = $01 {extended sound header}
cmpSH = $02 {compressed sound header}
```

The extended sample header (`extSH`) is the in-memory implementation of the Audio Interchange File Format standard expected by the Sound Manager. The AIFF standard specifies up to 32 bit sample sizes, up to 128 channels per file, and much more. Refer to the AIFF documentation for more details. The compressed sample header (`cmpSH`) is the compressed sample counter-part of the extended sample header. Refer to the Macintosh Audio Compression and Expansion documentation for further information.

Developers are free to use their own `encode` options with values in the range 64-127. Apple reserves the values 0 - 63.

The `baseNote` is the pitch at which the original sample was taken. If a harpsichord were sampled while playing middle C, then the `baseNote` is

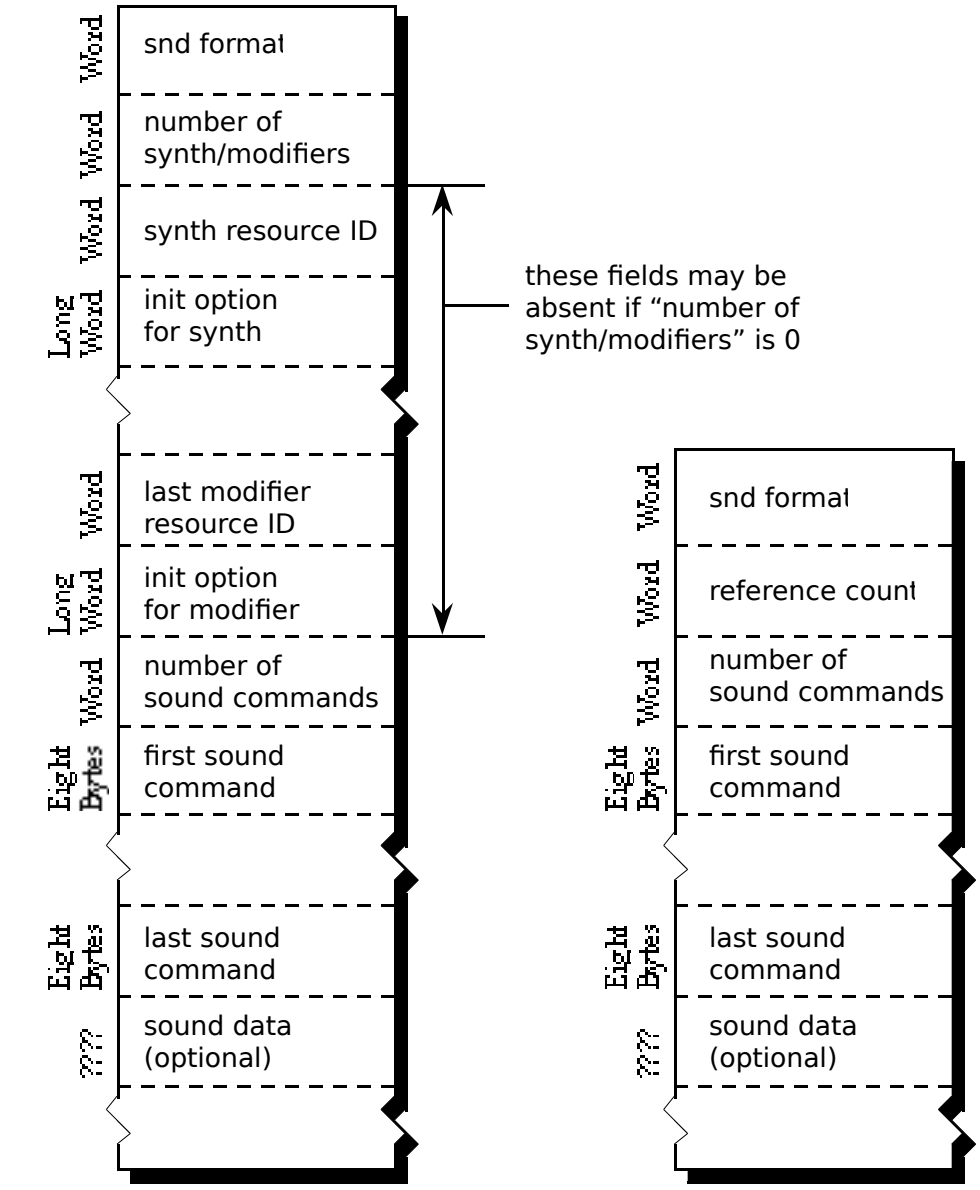
middle C. The baseNote values are 1 through 127 inclusive. (Refer to Table 4.) The baseNote allows the Sound Manager to calculate the proper play back rate of the sample when an application uses the noteCmd. Applications should not modify the baseNote of a sampled sound. To use the sample at different pitches, send the noteCmd or freqCmd.

Refer to the section “Current Sound Manager” regarding limitations with the noteCmd and freqCmd.

Each byte in the sampleArea data is similar in value to those in a wave table description. Each byte is a value of $00 through $FF inclusive; $80 represents a zero level of amplitude, $00 is the largest negative value, and $FF is the largest positive value.

The Sound Manager Summary contains the description of the data format to be used with 16 bit sampled sounds. Developers wishing to write custom synthesizers for their hardware are encouraged to use this data format. This data structure is intended to complement the use of the AIFF standard.

The 'snd ' Resource



'snd ' Format 1 'snd ' Format 2

Figure 3 'snd ' Resource Layout

Sound resources are intended to be simple, portable, and dynamic solutions for incorporating sounds into applications. Creating these `'snd '` or sound resources, requires some understanding of sound synthesis to build a sampled sound header, wave table data, and sound commands. There are two types of `'snd '` resources, format 1 and format 2. Figure 3 compares the structures of both of these formats. These resources should have their purgeable bit set or the application will need to call `_HPurge` after using the `'snd '`.

The format 1 `'snd '` was developed for use with the Sound Manager. A format 1 `'snd '` may be a sequence of commands describing a melody without specifying a synthesizer or modifier and without sound data. This would allow an application to use the `_SndPlay` routine on any channel to play that melody. A format 1 `'snd '` resource may contain a sampled sound or wave table data.

The format 2 `'snd '` was developed for use with HyperCard. It is intended for use with the sampled sound synthesizer only. A format 2 simply contains a sound command that points to a sampled sound header.

HyperCard (versions 1.2.1 and earlier) contain `'snd '` resources incorrectly labeled as format 1. Refer to Macintosh Technical Note #168.

Numbers for `'snd '` resources in the range 0 through 8191 are reserved for Apple. The `'snd '` resources numbered 1 through 4 are defined to be the standard system beep.

A sound command contained in a `'snd '` resource with associated sound data is marked by setting the high bit of the command. This changes the `param2` field of the command to be an offset value from the resource's beginning, pointing to the location of the sound data. Refer to Figure 5 showing the structure of a sound command. To calculate this offset, use one of the following formulas below.

For a format 1 `'snd '` resource, the offset is calculated as follows:

```
offset = 4 + (number of synth/mods * 6) + (number of cmds * 8)
```

For a format 2 `'snd '` resource, the offset is calculated as follows:

```
offset = 6 + (number of cmds * 8)
```

The first few bytes of the resource contain 'snd ' header information and are a different size for either format. Each synthesizer or modifier specified in a format 1 'snd ' requires 6 bytes. The number of synthesizers and/or modifiers multiplied by 6 is added to this offset. The number of commands multiplied by 8 bytes, the size of a sound command, is added to the offset.

Format 1 'snd ' Resource

Figure 3 shows the fields of a format 1 'snd ' resource. This resource may also contain the actual sound data for the wave table synthesizer or the sampled sound synthesizer. The number of synthesizer and modifiers to be used by this 'snd ' is specified in the field number of synth/modifiers. The synthesizer required to produce the sound described in the 'snd ' is specified by the field synth resource ID. If any modifiers are to be installed, their resource IDs follow the first synthesizer. Any synthesizer or modifier specified beyond this first one will be installed into the channel as a modifier.

For every synthesizer and modifier, an init option can be supplied in the field immediately following the resource ID for each synthesizer or modifier. The number of commands within the resource is specified in the field number of sound commands. Each sound command follows in the order they should be sent to the channel. If a command such as a bufferCmd is contained in this resource, it needs to specify where in the resource the sampled sound header is located. This is done by setting the high bit of the bufferCmd and supplying the offset in param2. Refer to the section “Sound Manager Commands”.

The 'snd ' resource may be only a sequence of commands describing a melody playable by any synthesizer. This allows the 'snd ' to be used on any channel. In this case the number of synth/modifiers should be 0, and there would not be a synth resource ID nor init option in the 'snd '.

Example Format 1 `'snd '`

The following example resource contains the proper information to create a sound with `_SndPlay` and the sampled sound synthesizer.

HEX	Size	Meaning
{beginning of snd resource, header information}		
$0001	WORD	format 1 resource
$0001	WORD	number of synth/modifiers to be installed
{synth ID to be used}		
$0005	WORD	resource ID of the first synth/modifier
$0000 0000	LONG	initialization option for first synth/modifier
$0001	WORD	number of sound commands to follow
{first command, 8 bytes in length}		
$8051	WORD	bufferCmd, high bit on to indicate sound data included
$0000	WORD	bufferCmd param1
$0000 0014	LONG	bufferCmd param2, offset to sound header (20 bytes)
{sampled sound header used in a soundCmd and bufferCmd}		
$0000 0000	LONG	pointer to data (it follows immediately}
$0000 0BB8	LONG	number of samples in bytes (3000 samples)
$56EE 8BA3	LONG	sampling rate of this sound (22kHz)
$0000 07D0	LONG	starting of the sample’s loop point
$0000 0898	LONG	ending of the sample’s loop point
$00	BYTE	standard sample encoding
$3C	BYTE	baseNote (middle C) at which sample was taken
{Packed Array [1..3000] OF Byte, the sampled sound data}		
$8080 8182 8487 9384 6F68 6D65 727B 8288		
$918E 8D8F 867E 7C79 6F6D 7170 7079 7F81		
$898F 8D8B...		

Format 2 'snd ' Resource

The format 2 'snd ' resource is used by the sampled sound synthesizer only and must contain a sampled sound. The _SndPlay routine supports this format by automatically opening a channel to the sample sound synthesizer and using the bufferCmd.

Figure 3 shows the fields of a format 2 'snd ' resource. The field reference count is for the application's use and is not used by the Sound Manager. The fields number of sound commands and the sound commands are the same as described in a format 1 resource. The last field of this 'snd ' is for the sampled sound. The first command should be either a soundCmd or bufferCmd with the pointer bit set in the command to specify the location of this sampled sound header. Any other sound commands in this 'snd ' will be ignored by the Sound Manager.

Example Format 2 'snd '

The following example resource contains the proper information to create a sound with _SndPlay and the sampled sound synthesizer.

```
HEX          Size   Meaning

{beginning of 'snd ' resource, header information}
$0002        WORD   format 2 resource
$0000        WORD   reference count for application's use
$0001        WORD   number of sound commands to follow

{first command, 8 bytes in length}
$8051        WORD   bufferCmd, high bit on to indicate sound data included
$0000        WORD   bufferCmd param1
$0000 0014   LONG   bufferCmd param2, offset to sound header (20 bytes)

{sampled sound header used in a soundCmd and bufferCmd}
$0000 0000   LONG   pointer to data (it follows immediately}
$0000 0BB8   LONG   number of samples in bytes (3000 samples)
$56EE 8BA3   LONG   sampling rate of this sound (22kHz)
$0000 07D0   LONG   starting of the sample's loop point
$0000 0898   LONG   ending of the sample's loop point
$00          BYTE   standard sample encoding
$3C          BYTE   baseNote (middle C) at which sample was taken

{Packed Array [1..3000] OF Byte, the sampled sound data}
$8080 8182 8487 9384 6F68 6D65 727B 8288
$918E 8D8F 867E 7C79 6F6D 7170 7079 7F81
$898F 8D8B...
```

The 'snth' Resource

The 'snth' resources are the routines that get linked to a sound channel used to create sound. The calls to _SndPlay, _SndNewChannel, _SndAddModifier, and _SndControl are mapped with unique 'snth' resources based on the hardware present on each Macintosh. The Sound Manager first determines the type of Macintosh being used. Then, using the id specified in one of the four routines above, adds a constant to this id. For the Macintosh Plus and SE, a constant of $1000 is added to this id. For the Macintosh II, $800 is added to the id. If the mapped resource ID is not available, the Sound Manager will use the actual id value specified.

The 'snth' resource IDs in the range 0 through 255 inclusive are reserved for Apple within the 'snth' resource mapping range.

Resource ID	Synthesizer	Target Macintosh
$0001	noteSynth	general for any Macintosh
$0003	waveTableSynth	general for any Macintosh
$0005	sampledSynth	general for any Macintosh
$0006-$00FF	*reserved for Apple*	general for any Macintosh
$0100-$0799	*free for developers*	general for any Macintosh
$0801	noteSynth	Mac with Apple Sound Chip$0803
waveTableSynth		Mac with Apple Sound Chip
$0805	sampledSynth	Mac with Apple Sound Chip
$0806-$08FF	*reserved for Apple*	Mac with Apple Sound Chip
$0900-$0999	*free for developers*	Mac with Apple Sound Chip
$1001	noteSynth	Mac Plus and SE
$1003	waveTableSynth	Mac Plus and SE
$1005	sampledSynth	Mac Plus and SE
$1006-$10FF	*reserved for Apple*	Mac Plus and SE
$1100-$1199	*free for developers*	Mac Plus and SE

Table 2 Synthesizer Resource IDs

For example, if an application requested the sampled sound synthesizer while running on the Macintosh Plus, it uses the resource ID of 5 when calling _SndNewChannel. The Sound Manager will then open the 'snth' resource with the ID of $1005 since this synthesizer is specific to the Macintosh Plus. Table 2 lists the current synthesizers and the IDs used by each Macintosh.

Refer to the section “Current Sound Manager” regarding the Macintosh II 'snth' IDs.

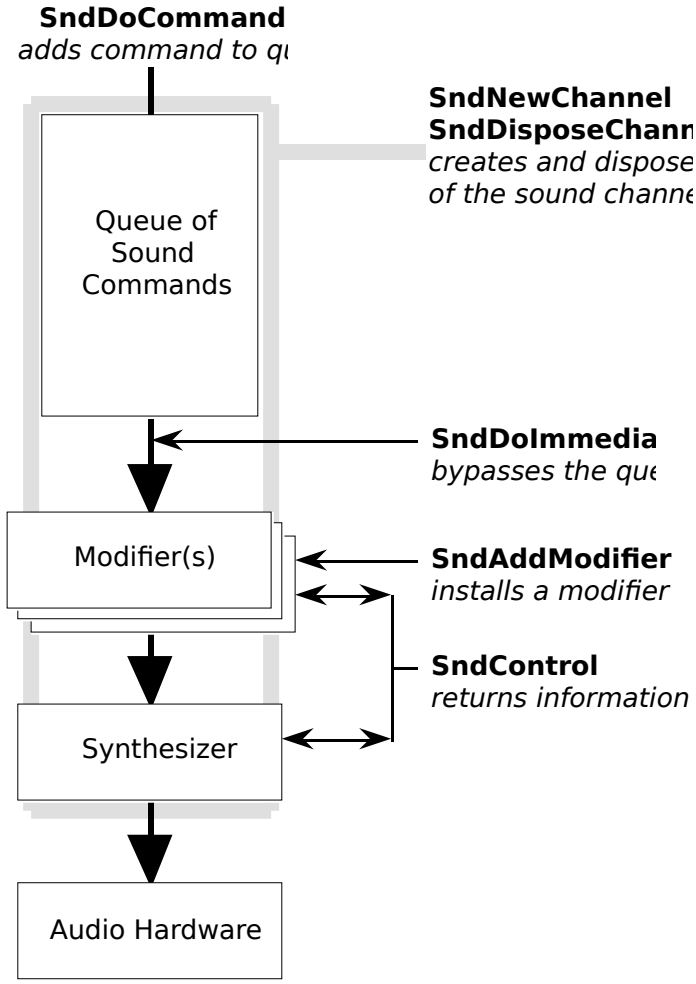


Figure 4 Sound Channel and Routines

```
FUNCTION SndPlay (chan: SndChannelPtr; sndHdl: Handle; async: BOOLEAN)
                  : OSErr;
```

The function `_SndPlay` is a higher level sound routine and is generally used separately from the other Sound Manager calls. `_SndPlay` will attempt to play the sound specified in the `'snd '` resource located at `sndHdl`. This is the only Sound Manager routine that accepts a `'snd '` resource as one of its parameters. If a format 1 `'snd '` specifies a synthesizer and any modifiers, those `'snth'` resource(s) will be loaded in memory and linked to the channel. All commands contained in the `'snd '` will be sent to the channel. If the application passes `NIL` as the channel pointer, `_SndPlay` will create a

channel in the application's heap. The Sound Manager will release this memory after the sound has completed. The `async` parameter is ignored if `NIL` is passed as the channel pointer.

If the application does supply a channel pointer in `chan`, the sound can be produced asynchronously. When sound is played asynchronously, a completion routine can be called when the last command has finished processing. This procedure is the `userRoutine` supplied with `_SndNewChannel`. `_SndPlay` will call `_HGetState` on the `'snd '` resource before `_HMoveHi` and `_HLock`, and once the sound has completed, will restore the state of the `'snd '` resource's handle with `_HSetState`.

If the format 1 `'snd '` resource does not specify which synthesizer is to be used, `_SndPlay` will default to the note synthesizer. `_SndPlay` will also support a format 2 `'snd '` resource using the sampled sound synthesizer and a `bufferCmd`. Note that a format 1 `'snd '` must use have a `bufferCmd` in order to be used with `_SndPlay` and the sampled sound synthesizer.

Do not use `_SndPlay` with a `'snd '` that specifies a synthesizer ID if the channel has already been linked to a synthesizer.

```
FUNCTION SndNewChannel (VAR chan: SndChannelPtr; synth: INTEGER;
                        init: LONGINT; userRoutine: ProcPtr) : OSErr;
```

When `NIL` is passed as the `chan` parameter, `_SndNewChannel` will allocate a sound channel record in the application's heap and return its `POINTER`. Applications concerned with memory management can allocate their own channel memory and pass this `POINTER` in the `chan` parameter. Typically this should not present a problem since a channel should only be in use temporarily. Each channel will hold 128 commands as a default size. The length of a channel can be expanded by the application creating its own channel in memory.

The `synth` parameter is used to specify which synthesizer is to be used. The application specifies a synthesizer by its resource ID, and this `'snth'` resource will be loaded and linked to the channel. The state of the `'snth'` handle will be saved with `_HGetState`. To create a channel without linking it with a synthesizer, pass 0 as the `synth`. This is useful when using `_SndPlay` with a `'snd '` that specifies a synthesizer ID.

The application may specify an `init` option that should be sent to the synthesizer when opening the channel. For example, to open the third wave table channel use `initChan2` as the `init`. Only the wave table synthesizer and sampled sound synthesizer currently use the `init` options. To determine if a particular option is available by the synthesizer, use the `availableCmd`.

```
initChanLeft      = $02; {left channel - sampleSynth only}
initChanRight     = $03; {right channel- sampleSynth only}
initChan0         = $04; {channel 1 - wave table only}
initChan1         = $05; {channel 2 - wave table only}
initChan2         = $06; {channel 3 - wave table only}
initChan3         = $07; {channel 4 - wave table only}
initSRate22k      = $20; {22k sampling rate - sampleSynth only}
initSRate44k      = $30; {44k sampling rate - sampleSynth only}
initMono          = $80; {monophonic channel - sampleSynth only}
initStereo        = $C0; {stereo channel - sampleSynth only}
```

Refer to the section “Current Sound Manager” regarding `init` options and the sampled sound synthesizer.

If an application is to produce sounds asynchronously or needs to be alerted when a command has completed, it uses a `CallBack` procedure. This routine will be called once the `callBackCmd` has been received by the synthesizer. If you pass `NIL` as the `userRoutine`, then any `callBack` command will be ignored.

```
FUNCTION SndAddModifier (chan: SndChannelPtr; modifier: ProcPtr;
                         id: INTEGER; init: LONGINT) : OSErr;
```

This routine is used to install a `modifier` into an open channel specified in `chan`. The `modifier` will be installed in front of the synthesizer or any existing modifiers in the channel. If the `modifier` is saved as a `'snth'` resource, pass `NIL` for the `ProcPtr` and specify its resource ID in the parameter `id`. This will cause the Sound Manager to load the `'snth'` resource, lock it in memory, and link it to the channel specified. The state of the `'snth'` resource handle will be saved with `_HGetState`. Refer to the section “User Routines” for more information regarding writing a modifier.

Refer to the section “Current Sound Manager” regarding `modifier` resources.

```
FUNCTION SndDoCommand (chan: SndChannelPtr; cmd: SndCommand;
                       noWait: BOOLEAN) : OSErr;
```

This routine will send the sound command specified in `cmd` to the existing channel's command queue. If the parameter `noWait` is set to `FALSE` and the queue is full, the Sound Manager will wait until there is space to add the command. If `noWait` is set to `TRUE` and the channel is full, the Sound Manager will not send the command and returns the error "`queueFull`".

```
FUNCTION SndDoImmediate (chan: SndChannelPtr; cmd: SndCommand): OSErr;
```

This routine will bypass the command queue of the existing channel and send the specified command directly to the synthesizer, or the first modifier. This routine will also override any `waitCmd`, `pauseCmd` or `syncCmd` that may have been received by the synthesizer or modifiers.

```
FUNCTION SndControl (id: INTEGER; VAR cmd: SndCommand) : OSErr;
```

This routine is used to send control commands directly to a synthesizer or modifier specified by its resource `ID`. This can be called even if no channel has been created for the synthesizer. This control call is used with the `availableCmd` or `versionCmd` to request information regarding a synthesizer. The result of this call is returned in `cmd`.

```
FUNCTION SndDisposeChannel (chan: SndChannelPtr; quietNow: BOOLEAN) :
                            OSErr;
```

This routine will dispose of the channel specified in `chan` and release all memory created by the Sound Manager. If an application created its own channel record in memory or installed a sound as an instrument, the Sound Manager will not dispose of that memory. The Sound Manager will restore the original state of `'snth'` resource handles with a call to `_HSetState`.

`_SndDisposeChannel` can either immediately dispose of a channel or wait until the queued commands are processed. If `quietNow` is set to `TRUE`, a `flushCmd` and then a `quietCmd` is sent to the channel. This will remove all commands, stop any sound in progress and close the channel. If `quietNow` is set to `FALSE`, then the Sound Manager will issue a `quietCmd` only and wait until the `quietCmd` is received by the synthesizer before disposing of the channel. In this situation `_SndDisposeChannel` will be synchronous.

Command Descriptions

Sound commands are placed into a channel one after the other. At the end of the channel is the synthesizer which interprets the command and plays the sound with the hardware. All synthesizers are designed to accept the most general set of sound commands. Some commands are specific to only a particular synthesizer. There are some commands and options that may not be currently implemented by a synthesizer. Refer to section “The Current Sound Manager” for more details.

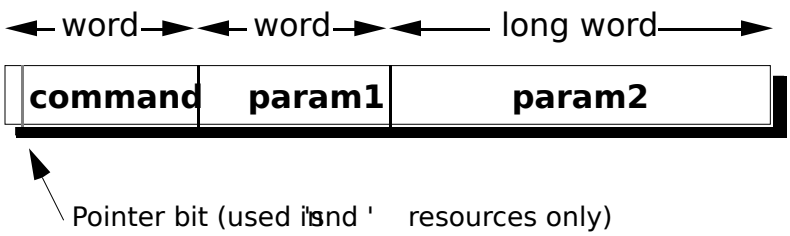


Figure 5 Generic Command Format

Figure 5 shows the structure of a generic sound command. Commands are always eight bytes in length. The first two bytes are the command number, and the next six make up the command’s options. The format of these last six bytes will depend on the command being used.

The `pointer bit` is only used by `'snd '` resources that contain commands and associated sound data (i.e. sampled sound or wave table data). If the high bit of the command is set, then `param2` is an offset specifying where the associated data is located. This offset is the number of bytes starting from the beginning of the resource to the associated sound data. The section “Sound Resources” shows how this offset is calculated.

cmd=nullCmd param1=0 param2=0

This command is sent by modifiers. It is simply absorbed by the Sound Manager and no action is performed. Modifiers use a `nullCmd` to replace commands in a

channel to prevent them from being sent to a synthesizer.

cmd=initCmd param1=0 param2=init

This command is only sent by the Sound Manager. It will send an `initCmd` to the synthesizer when an application uses the routines `_SndPlay`, `_SndNewChannel` or `_SndAddModifier`. This causes a synthesizer or modifier to allocate its private memory storage and to use the `init` option.

cmd=freeCmd param1=0 param2=0

This command is only sent by the Sound Manager. It is exactly opposite of the `initCmd`. When an application calls `_SndDisposeChannel`, the Sound Manager will send the `freeCmd` to the synthesizer. This causes the synthesizer to dispose of all the private memory it had allocated.

cmd=quietCmd param1=0 param2=0

This command is sent by an application using `_SndDoImmediate`. It will cause the synthesizer to stop any sound in progress. It is also sent by the Sound Manager with the `_SndDisposeChannel` routine.

cmd=flushCmd param1=0 param2=0

This command is sent by an application using `_SndDoImmediate`. It will cause all commands in the channel be be removed. It is also sent by the Sound Manager from `_SndDisposeChannel` when `quietNow` is `TRUE`.

cmd=waitCmd param1=duration param2=0

This command is sent by an application or a modifier. It will suspend all processing in the channel for the number of half-milliseconds specified in `duration`. A one second wait would be a `duration` of `2000`.

cmd=pauseCmd param1=0 param2=0

This command is sent by an application or a modifier to cause the channel to suspend processing until a `tickleCmd` or `resumeCmd` is received.

cmd=resumeCmd param1=0 param2=0

This command is sent by an application or a modifier to cause a channel to resume processing of commands. This is the opposite of the `pauseCmd`.

cmd=callBackCmd param1=*user-defined*
param2=*user-defined*

This command is sent by an application. The `callBackCmd` causes the Sound Manager to call the `userRoutine` specified in `_SndNewChannel`. The two parameters of this command can be used by the application for any purpose. This allows an application to have a general `userRoutine` for any channel. By using `param1` and `param2` with unique values, the `CallBack` procedure can test for specific actions to take. Refer to the section “User Routines”.

This command is used as a marker for an application to determine at what point the channel has reached in processing its queue. It is mostly used to determine when to dispose of a channel, since the `callBackCmd` is generally the last command sent. It can also be used to allow an application to synchronize sounds with other actions.

cmd=syncCmd param1=count param2=identifier

This command is sent by an application. Every `syncCmd` is held in the channel, suspending any further processing until its `count` equals 0. The Sound Manager will first decrement the `count` and then wait for another `syncCmd` having the same `identifier` to be received on another channel.

To synchronize four wave table channels, send the `syncCmd` to each channel with `count = 4` giving each command the same `identifier`. If a channel should wait for two more `syncCmds`, then its `count` would be `3`. If a channel is to wait for one more `syncCmd`, its `count` would be sent as `2`.

Refer to the section “Current Sound Manager” regarding the `count` parameter of a `syncCmd`.

cmd=emptyCmd param1=0 param2=0

This command is only sent by the Sound Manager. Synthesizers expect to receive additional commands after a `resumeCmd`. If no other commands are to be sent, the Sound Manager will send an `emptyCmd`.

cmd=tickleCmdparam1=0 param2=0

This command is only sent by the Sound Manager to a modifier. This will cause modifiers to perform their requested periodic actions. If the `tickleCmd` had been requested by a `howOftenCmd`, then a `tickleCmd` will be sent periodically according to the `period` specified in the `howOftenCmd`. If the `tickleCmd` had been requested by an `wakeUpCmd`, then this command will be

sent only once according to the period specified in the wakeUpCmd. A tickleCmd command will also resume a channel suspended by a pauseCmd.

cmd=requestNextCmd param1=count param2=0

This command is only sent by the Sound Manager in response to a modifier returning TRUE. Refer to the section "User Routine" discussing modifiers. Count is the number of consecutive times that the modifier has requested another command.

cmd=howOftenCmd param1=period param2=pointer

This command is sent by a modifier and will instruct the Sound Manager to periodically send a tickleCmd. Param1 contains the period (in half-milliseconds) that a tickleCmd should be sent. Param2 contains a POINTER to the modifier stub.

cmd=wakeUpCmd param1=period param2=pointer

This command is sent by a modifier and will instruct the Sound Manager to send a single tickleCmd after the period specified (in half-milliseconds). Param2 contains a POINTER to the modifier stub.

The howOftenCmd and the wakeUpCmd are mutually exclusive. Sending one will cancel the other.

cmd=availableCmd param1=result param2=init

This command is sent by an application to determine if certain characteristics specified in the init parameter are available from the synthesizer. This command can only be used with the _SndControl routine. These init options are documented under the _SndNewChannel routine and are passed in param2 of the availableCmd.

```
myCmd.cmd := availableCmd;
myCmd.param1 := 0;
myCmd.param2 := initStereo;   {we'll test for a stereo channel}
myErr := SndControl (sampledSynth, myCmd);
IF (myCmd.param1 <> 0) THEN stereoAvailable := TRUE;
```

The result is returned in param1. A result of 1 is returned if the synthesizer has the requested characteristics. If it does not, the result is 0.

Refer to section "Current Sound Manager" regarding limitations with the availableCmd.

cmd=versionCmd param1=0 param2=version

This command is sent by applications and the Sound Manager to determine which version of the synthesizer is available. The `versionCmd` can only be sent with the `_SndControl` routine. The `version` is returned in `param2`. Version 1.2 of a synthesizer would be returned as `$0001 0002`.

cmd=noteCmd param1=duration
param2=amplitude + frequency

This command is sent by applications and modifiers to specify a note for either the note synthesizer, or with an instrument installed into the channel. The `duration` parameter is in half-milliseconds. A `duration` of `2000` would be a duration of one second. The maximum `duration` is a `duration` of `32767` or about 16 seconds. The structure of a `noteCmd` is given in Figure 6.

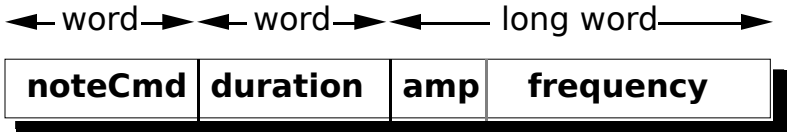


Figure 6 noteCmd Format

The `param2` of a `noteCmd` is a combination of an `amplitude` and a `frequency`. The `amplitude` is passed in the high byte and the lower three bytes are the `frequency`. The `frequency` can be specified in two ways, as a decimal note (refer to the section “Note Values and Durations”) or a frequency value (refer to `freqCmd`). The `amplitude` values range from `$00` to `$FF` inclusively. The following example demonstrates the use of a `noteCmd`.

```
amp := $FF000000;              {loudest possible amplitude}
note := 60;                    {middle C}
myCmd.cmd := noteCmd;
myCmd.param1 := 2000;          {one second duration}
myCmd.param2 := amp + note;
myErr := SndDoCommand(myChan, myCmd, FALSE);
```

The `noteCmd` will start at the beginning of a sampled sound. The `noteCmd` uses the loop points of the header to extend the length of the sound to the `duration` specified in a `noteCmd`. There must be a loop ending point specified in the header in order for the `noteCmd` to work properly.

Refer to the section “Current Sound Manager” regarding limitations with the `noteCmd` and using `amplitude`.

cmd=restCmd param1=duration param2=0

This command is sent by applications and modifiers to cause the channel to rest for the `duration` specified in half-milliseconds.

cmd=freqCmd param1=0 param2=frequency

This command is sent by applications and modifiers. A `frequency` can be sent to a synthesizer to change the pitch of a sound. It is similar to the `noteCmd` in that a decimal note value can be used instead of a frequency value. The structure of this command is shown in Figure 7. If no sound is playing, it causes the synthesizer to begin playing at the specified `frequency` for an indefinite duration. The upper byte of `param2` is ignored. A frequency value is sent in the lower three bytes of `param2`, where the frequency desired is multiplied by 256. For example, to specify a frequency of 440 Hz (the A below middle C) the `frequency` value would be `440 * 256` or `112640`.

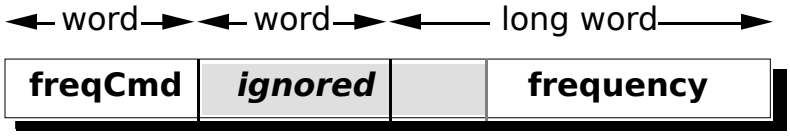


Figure 7 freqCmd format

Refer to the section “Current Sound Manager” regarding the limitations of the `freqCmd`.

cmd=ampCmd param1=amplitude param2=0

This command is sent by applications and modifiers to change the `amplitude` of the sound in progress. If no sound is currently playing, then it will affect the `amplitude` of the next sound.

Refer to the section “Current Sound Manager” regarding the use of `amplitude`.

cmd=timbreCmd param1=timbre param2=0

This command is sent by applications and modifiers. It is used only by the note synthesizer to change its timbre or tone. A sine wave is specified as `0` in

param1 and produces a flute-like sound. A value of 255 in param1 represents a modified square wave and produces a buzzing or reed-like sound. Changing the note synthesizer's timbre should be done before playing the sound. Only a Macintosh with the Apple Sound Chip will allow this command to be sent while a sound is in progress.

cmd=waveTableCmd param1=length param2=pointer

This command is sent by applications. It is only used by the wave table synthesizer. It will install a wave table to be used as an instrument by supplying a POINTER to the wave table in param2.

All wave cycles will be re-sampled to 512 bytes.

cmd=phaseCmd param1=shift param2=pointer

This command is sent by applications. It is only used by the wave table synthesizer to synchronize the phases of the wave cycles across different wave table channels. As an example, if two wave table channels containing the same wave cycle were sent the same noteCmd, they could not begin exactly at the same time. Therefore, to synchronize the wave cycles for these two channels the phaseCmd is sent.

This prevents the phasing effects of playing two similar waves together at the same pitch. The channel will have its wave shifted by the amount specified in shift to correspond with the wave's phase in the channel specified in param2. The shift value is a 16 bit fraction going from zero to one. The value of $8000 would be the half-way point of the wave cycle. Generally, the effects from this command will not be noticed.

Refer to the section "Current Sound Manager" regarding the phaseCmd.

cmd=soundCmd param1=0 param2=pointer

This command is sent by an application and is only used by the sampled sound synthesizer. If the application sends this command, param2 is a POINTER to the sampled sound locked in memory. The format of a sampled sound is shown in section "The Sampled Sound Synthesizer". This command will install the sampled sound as an instrument for the channel. If the soundCmd is contained within a 'snd ' resource, the high bit of the command must be set. To use a sampled sound 'snd ' as an instrument , first obtain a POINTER to the sampled sound header locked in memory. Then pass

this POINTER in param2 of a soundCmd. After using the sound, the application is expected to unlock this resource and allow it to be purged.

cmd=bufferCmd param1=0 param2=pointer

This command is sent by applications and the Sound Manager to play a sampled sound, in one-shot mode, without any looping. The POINTER in param2 is the location of a sampled sound header locked in memory. The format of a sampled sound is shown in section "The Sampled Sound Synthesizer". A bufferCmd will be queued in the channel until the preceding commands have been processed. If the bufferCmd is contained within a 'snd ' resource, the high bit of the command must be set. If the sound was loaded in from a 'snd ' resource, the application is expected to unlock this resource and allow it to be purged after using it.

Refer to the section "Current Sound Manager" regarding the bufferCmd.

cmd=rateCmd param1=0 param2=rate

This command is sent by applications to modify the pitch of the sampled sound currently playing. The current pitch is multiplied by the rate in param2. It is used for pitch bending effects. The default rate of a channel is 1.0. To cause the pitch to fall an octave (or half of its frequency), send the rateCmd with param2 equal to one half as shown below.

```
myCmd.cmd := rateCmd;
myCmd.param1 := 0;
myCmd.param2 := FixedRatio(1, 2);
myErr := SndDoImmediate(myChan, myCmd);
```

cmd=continueCmd param1=0 param2=pointer

This command is sent by applications to the sampled sound synthesizer. It is similar to the bufferCmd. Long sampled sounds may be broken up into smaller sections. In this case, the application would use the bufferCmd for the first portion and the continueCmd for any remaining portions. This will result in a single continuous sound with the first byte of the sample being joined with the last byte of the previous sound header without audible clicks.

Refer to the section "Current Sound Manager" regarding the continueCmd.

USER ROUTINES

These user routines will be called at interrupt time and therefore must not attempt to allocate, move or dispose of memory, de-reference an unlocked handle, or call other routines that do so. Assembly language programmers must preserve all registers other than A0-A1, and D0-D2. If these routines are to use an application's global data storage, it must first reset A5 to the application's A5 and then restore it upon exit. Refer to Macintosh Technical Note #208 regarding setting up A5.

```
PROCEDURE CallBack(chan: SndChannelPtr; cmd: SndCommand);
```

The function _SndNewChannel allows a completion routine or CallBack procedure to be associated with a channel. This procedure will be called when a callBackCmd is received by the synthesizer linked to that channel. This procedure can be used for various purposes. Generally it is used by an application to determine that the channel has completed its commands and to dispose of the channel. The CallBack procedure itself cannot be used to dispose of the channel, since it may be called at interrupt time.

A CallBack procedure can also be used to signal that a channel has reached a certain point in the queue. An application may wish to perform particular actions based on how far along the sequence of commands a channel has processed. Applications can use param1 or param2 of the callBackCmd as flags. Based on certain flags for certain channels, the call back can perform many different functions. The CallBack procedure will be passed the channel that received the callBackCmd. The entire callBack command is also passed to the CallBack procedure.

```
myCmd.cmd := callBackCmd;     {install the callBack command}
myCmd.param1 := 0;            {not used in this example}
myCmd.param2 := SetCurrentA5; {pass the callBack our A5}
myErr := SndDoCommand (myChan, myCmd, FALSE);
```

The example code above is used to setup a callBackCmd. Note that param2 of a sound command is a LONGINT. This can be used to pass in the application's A5 to the CallBack procedure. Once this command is received by the synthesizer, the following example CallBack procedure can set A5 in order to access the application's globals. The function's SetCurrentA5 and SetA5 are documented in Macintosh Technical Note #208.

```
Procedure SampleCallBack (theChan: SndChannelPtr; theCmd: SndCommand);

VAR
  theA5 : LONGINT;

BEGIN
  theA5 := SetA5(myCmd.param2);     {set A5 and get current A5}
  callBackPerformed := TRUE;        {global flag}
  theA5 := SetA5(theA5);            {restore the current A5}
END;


FUNCTION Modifier(chan: SndChannelPtr; VAR cmd: SndCommand;
                        mod: ModifierStubPtr) : BOOLEAN
```

A modifier will be called when the command reaches the end of the queue, before being sent to the synthesizer or other modifiers that may be installed. `Chan` will contain the channel pointer allowing multiple wave table channels to be supported by the same modifier. The `ModifierStub` is a record created by the Sound Manager during the call `_SndAddModifier`. A pointer to the `ModifierStub` is in `mod`. There are two special commands that the modifier must support, the `initCmd` and the `freeCmd`.

Refer to the section “Current Sound Manager” regarding `modifiers` being saved as resources.

```
ModifierStub = PACKED RECORD
               nextStub:      ModifierStubPtr; {pointer to next stub}
               code:          ProcPtr;    {pointer to modifier}
               userInfo:      LONGINT;    {free for modifier's use}
               count:         Time;       {used internally}
               every:         Time;       {used internally}
               flags:         SignedByte; {used internally}
               hState:        SignedByte; {used internally}
            END;
```

The `initCmd` is sent by the Sound Manager when an application calls `_SndAddModifier`. This is a command telling the modifier to allocate any additional data. The `ModiferStub` contains a four byte field, `userInfo`, that can be used as a pointer to this additional memory. The `initCmd` will not be sent to a `modifier` at interrupt time. This allows a `modifier` to allocate memory and save the current application’s `A5`. All memory storage allocated by the modifier must be locked, since the modifier will be called at interrupt time.

The `freeCmd` will be sent to the modifier when the Sound Manager is disposing of the channel. This command will not be sent at interrupt time. At this point the modifier should free any data it may have allocated.

A modifier will be given the current command, before the command is sent to the synthesizer or other modifiers. The current command is sent to the modifier in the variable `cmd`. A `nullCmd` is never sent to a modifier. If the modifier wished to ignore the current command and allow it to be sent on, it would return `FALSE`. To remove the current command, replace it with a `nullCmd` and then return `FALSE`. To alter the current command, replace it with the new one and return `FALSE`. Returning `FALSE` means that the modifier has completed its function.

If the modifier is to send additional commands to the channel, the function will return `TRUE` and may or may not change the current command. The Sound Manager will call the modifier again sending it a `requestNextCmd`. The modifier can then replace this command with the one desired. The modifier can continue to return `TRUE` to send additional commands. The `requestNextCmd` will indicate the number of times this command has been consecutively sent to the modifier.

Modifiers are short routines used to perform real-time modifications on channels. Having too many modifiers, or a lengthy one, may degrade performance.

Synthesizer Details

This section documents the details for each of the current synthesizers.

The Note Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.
- Commands currently supported:

```
  availableCmd  versionCmd      freqCmd
noteCmdrestCmd    flushCmd
quietCmdampCmd   timbreCmd
```

Limitations of the Note Synthesizer

- Amplitude change is only supported by a Macintosh with the Apple Sound Chip, and is not supported by a Macintosh Plus or Macintosh SE.
- Only a single monophonic channel can be used.

The Wave Table Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.
- Commands currently supported:

```
  availableCmd  versionCmd      freqCmd
noteCmdrestCmd    flushCmd
quietCmdwaveTableCmd
```

Limitations of the Wave Table Synthesizer

- This synthesizer is not functioning on a Macintosh Plus or

Macintosh SE.

- A maximum of four channels can be open at any time.
- Amplitude change is not supported on any Macintosh.

- The one-shot mode is not supported on any Macintosh.
- The `phaseCmd` is not working.

The Sampled Sound Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.
- Commands currently supported:

```
  availableCmd  versionCmd      freqCmd
noteCmdrestCmd    flushCmd
quietCmdrateCmd   soundCmd
bufferCmd
```

Limitations of the Sampled Sound Synthesizer

- Amplitude change is not supported on any Macintosh.
- The current hardware will only support sampling rates up to 22kHz. This is not a limitation to the playback rates, and samples can be pitched higher on playback.
- There can only be a single monophonic channel; stereo is not supported.
- The `continueCmd` is not working.

The MIDI Synthesizer

- The version shipped with System 6.0.2 is `$0001 0002`.

Limitations of the MIDI Synthesizer

- The `midiDataCmd` documented in *Inside Macintosh Volume V* cannot be used.
- Fully functional MIDI applications cannot be written using the current Sound Manager and were intended as a

“poor man’s” method of sending notes to a MIDI keyboard.

- A bug in the MIDI synthesizer code prevents it from working after calling `_SndDisposeChannel`.

Sound Manager Bugs

This is a list of all known bugs and possible work-arounds in the System 6.0.2 Sound Manager. Each of these issues are being addressed and are expected to be solved with the next Sound Manager release.

Macintosh II `'snth'` IDs

The System 6.0.2 `'snth'` resources for the Macintosh II are incorrectly numbered. They should be `$0801-$0805`, but were shipped as `$0001-$0005`. This does not currently present a problem for applications, since the Sound Manager will default to these versions while running on the Macintosh II.

availableCmd

The `availableCmd` is returning a value of `1`, meaning `TRUE`, even if the synthesizer is actually no longer available. For example, after calling `_SndNewChannel` for the `noteSynth`, the `availableCmd` for the `noteSynth` should return `FALSE` since there isn't a second one. Furthermore, considering that only one synthesizer can be active at one time, after opening the `noteSynth` the `sampledSynth` is not available, but this command reports that it is. The only time the `availableCmd` will return `FALSE` is by requesting an `init` option that a synthesizer doesn't support, such as stereo channels.

_SndAddModifier

A modifier resource used in multiple channels must be pre-loaded and locked in memory by the application. There is a bug when the Sound Manager is disposing of a channel causing the modifier to be unlocked, regardless of other channels that may be using that modifier. If the application locks the modifier before installing it in the channel, the Sound Manager will not unlock it, but restores its state with `_HSetState`.

syncCmd

This command has a bug causing the `count` to be decremented incorrectly. To synchronize four channels, the same `count = 4` should be sent to all channels. The bug is with the Sound Manager decrementing all of the `count` values with every new `syncCmd`. In order to work around this, an application can synchronize four wave table channels by sending the `syncCmd` with `count = 4`. Then a `syncCmd` with the same `identifier` is sent to the second channel, this time with `count = 3`. The third channel is sent a `syncCmd` with `count = 2`. Finally, the last channel is sent with the `count = 1`. As

soon as the fourth `syncCmd` is received, all channels will have their `count` at 0 and will resume processing their queued commands. This bug will be fixed eventually, so test for the version of the synthesizer being used before relying on this.

`bufferCmd`

Sending a `bufferCmd` will reset the channel's `amplitude` and `rate` settings. Since the `amplitude` is already being ignored and the `rate` isn't typically used, this problem is not of much concern at this time.

`noteCmd`

This command may cause the sampled sound synthesizer to loop until another command is sent to the channel. This occurs when using a sampled sound installed as an instrument. If a `noteCmd` is the last command in the channel, the sound will loop endlessly. The work-around is to send a command after the final `noteCmd`. A `callBackCmd`, `restCmd` or `quietCmd` would be good.

`noteCmd` and **`freqCmd`**

These commands currently only support note values 1 through 127 inclusive. Refer to Table 4 for these values.

`_SysBeep`

On a Macintosh Plus or SE (which do not have the Apple Sound Chip) the Sound Manager will purge the application's channel of its `'snth'` or sound data. The application would have to dispose of the channel at this point and recreate a new one. This is another reason to release channels as soon as the application has completed its sound. This bug can be avoided by selecting the "Simple Beep" in the Control Panel's sound `'cdev'`. Applications should dispose of all channels before allowing a `_SysBeep` to occur. This includes putting up an alert or modal dialog that could cause the system beep. Since a foreground application under MultiFinder could cause a `_SysBeep` while the sound application is in the background, all applications should dispose of channels at a suspend event.

SOUND MANAGER ABUSE

Sound channels are for temporary use, and should only be created just before playing sound. Once the sound is completed, the channel should be disposed. Applications should not hold on to these channels for extended periods. The amount of overhead in `_SndNewChannel` is minimal. Basically, it is only a Memory Manager call. As long as the application holds onto a channel linked to a synthesizer, the `_SysBeep` call will not work and may cause trouble for the application's channel.

Friendly applications will dispose of all open channels during a suspend event from MultiFinder. If an application created a channel and then gets sent into the background, any foreground application or `_SysBeep` will be unable to gain access to the sound hardware.

Applications must dispose of all channels before calling `_ExitToShell`. Currently, calling `_ExitToShell` while generating a sound on the Macintosh Plus and SE will cause a system crash. So, calling `_SndDisposeChannel` before `_ExitToShell` will solve this issue. Setting `quietNow` to be `FALSE` will allow the application to complete the sound before continuing.

Do not mix older Sound Driver calls with the newer Sound Manager routines. The older Sound Driver should no longer be used. The Sound Manager is its replacement, providing all of it predecessor's abilities and more. Note that `_GetSoundVol` and `_SetSoundVol` are not part of the Sound Manager. They are used for setting parameter RAM, not the amplitude of a channel. Support for the older Sound Driver may eventually be discontinued.

The `'snd '` resource is so flexible that a warning of resource usage is needed. Most of the problems developers have with the Sound Manager are related more to the `'snd '` being used and less to the actual routines. Editing and creating `'snd '` resources with ResEdit is difficult. Many of the issues required in dealing with a `'snd '` are not supported by third party utilities. It is best to limit the `'snd '` to contain either sound data (i.e. sample sound) or a sequence of sound commands. Do not attempt to create resources that contain multiple sets of sound data.

Be very careful with what `'snd '` resources the application is intending to support. Test for the proper format and proper fields beforehand. An application needs to know the exact contents of the entire `'snd '` in order to

properly handle it. Things can get ugly real quick considering variant records, variable record lengths, and the pointer math that will be required.

If an application wants to use `_SndPlay` with an existing channel already linked to a synthesizer, the `'snd '` must not contain any `synth` information. With a format 1 `'snd '`, the `number of synth/modifiers` field must be `0`, and no `synth ID` or `init option` should be in the resource. Applications can only call `_SndPlay` with a channel linked to a synthesizer using a format 1 `'snd '` that contains sound commands without `synth` information.

A format 2 `'snd '` can never be used with `_SndPlay` more than once with an existing channel. This `'snd '` is assumed to be for the sampled sound synthesizer and `_SndPlay` will link this synthesizer to the channel. If a channel is created before calling `_SndPlay` with a format 2, specify `synth = 0` in the call to `_SndNewChannel`. After calling `_SndPlay` once, the application will have to dispose of the channel before using a format 2 `'snd '` again.

FREQUENTLY ASKED QUESTIONS

Q: Is there a way to determine if a sound is being made?

A: It is not possible at this time to determine if a synthesizer is currently active or producing a sound. However, an application can use the `callBackCmd` to determine when a sound has completed.

Q: How do I determine if the Apple Sound Chip is present?

A: There is no supported method for determining this. A new `_SysEnvirons` record is being considered to contain this information.

Q: How can I use the Sound Manager for a metronome effect?

A: Use a modifier to send a `noteCmd` to the note synthesizer. The modifier will use the `howOftenCmd` to cause the Sound Manager to send a `tickleCmd`. Every time the modifier gets called, it can send a

`noteCmd` to cause the click.

Q: What is the maximum number of synthesizers that can be opened at once? Can I have the `noteSynth` and the `sampledSynth` open at the same time and produce sound from either?

A: Only one synthesizer can be active at any time. This is because the active synthesizer “owns” the sound hardware until the channel is disposed of.

Q: How can I tell if more than four wave table channels are open or if another application has already open a synthesizer?

A: It is not possible at this time to determine when more than the maximum number of wave table channels has been allocated due to a limitation with the `availableCmd`. This issue is being investigated. It is not possible to determine if a synthesizer is in use by another application. If all applications would dispose of their channels at the resume event, this would not be a problem.

Q: How do I get `_SndPlay` to play the sound asynchronously? The Sound Manager seems to ignore the `async` parameter.

A: If `NIL` is used for the channel, then `_SndPlay` does ignore the `async` flag. To play the sound asynchronously, create a new channel with `_SndNewChannel` and pass this channel's pointer to `_SndPlay`. Again, if this 'snd ' contains `'snth'` information you must not link a synthesizer to the channel. Pass `0` as the `synth` in the call to `_SndNewChannel`.

Q: Should we use `'snd '` format 1 or format 2 for creating sound resources?

A: The format 1 `'snd '` is much more versatile. It can be used in the `_SndPlay` routine for any synthesizer and requires minimal programming effort. There is no recommendation for using either format. A format 1 has more advantages, and may contain everything a format 2 does. A format 2 is for a sampled sound only.

Q: I've opened a channel for the sampled sound synthesizer and I'm using `_SndPlay`. After awhile the system either hangs or crashes. What's wrong?

A: This is the most common abuse of the Sound Manager. The `'snd '` being used has specified a `'snth'` resource (a format 2 `'snd '` is assumed for the sampled sound synthesizer). The Sound Manager will attempt to link this `'snth'` to the channel with every call to `_SndPlay`. What's wrong is that the synthesizer has already been installed and the Sound Manager is attempting to install it again, only this time as a modifier. The same `'snth'` code has been install more

than once in the channel. If the 'snd ' contains 'snth' information, then _SndPlay can be used once and only once on a channel. There two possible solutions: Do the pointer math to obtain the sampled sound header and use the bufferCmd, or dispose of the channel after each call to _SndPlay.

Q: How can I use a sampled sound to play a sequence of notes?

A: Begin by opening a sampled sound channel. Load and lock the `'snd '` resource containing the sample sound into memory. Then obtain a pointer to the sampled sound header. Pass this pointer to the channel using the `soundCmd`. Now the sound is installed and ready for a sequence of `noteCmd`s. This sampled sound must contain an ending loop point or the `noteCmd` may not be heard.

Q: How do I change the play back rate of a sampled sound? Do I use the `freqCmd` or the `rateCmd`?

A: It is possible to change the `sampling rate` contained in the sampled sound header and then use the `bufferCmd`. The `freqCmd` currently requires decimal note values and will not support real frequency values. The `rateCmd` will only affect a sound that is currently in progress and is used for pitch bending effects. It is possible to add a few bytes of silence to the beginning of the sample to allow the `rateCmd` enough time to adjust the play back rate without hearing the bending affect on its pitch.

Q: How can I play multiple sampled sounds to play as a single sampled sound without the glitch that is heard between each sample on the Mac Plus?

A: On the Macintosh Plus or SE, the Sound Manager uses a 370 byte buffer internally to play sampled sounds. If the array of sampled sound data is in multiples of 370 bytes, the Sound Manager will not have to pad its internal buffer with silence. Using double buffering techniques, an application can send multiple sampled sounds using the `bufferCmd` from a `CallBack` procedure to create a continuous sound. Use this technique until the `continueCmd` is supported.

Q: How can I use the MIDI synthesizers with my own keyboards?

A: They have too many limitations at this time. Don’t bother trying.

	Tempo in beats/min	30	60	90	120	150	180
w	**whole note**	16000	8000	5333	4000	3200	2667
h	**half note**	8000	4000	2667	2000	1600	1333
q.	**dotted quarter note**	6000	3000	2000	1500	1200	1000
q	**quarter note**	4000	2000	1333	1000	800	667
e.	**dotted eighth note**	3000	1500	1000	750	600	500
e	**eighth note**	2000	1000	667	500	400	333
x.	**dotted sixteenth note**	1500	750	500	375	300	250
x	**sixteenth note**	1000	500	333	250	200	167

Table 3 duration values

Table 3 shows the duration values that are used in a `waitCmd`, `howOftenCmd`, `wakeUpCmd`, `noteCmd`, and `restCmd`. Their `duration` is in half-millisecond values. This chart will help in determining the actual `duration` used in certain tempos. To calculate the `duration` use the following formula.

```
duration = (2000/(beats per minute/60)) *
beats per note
```

To calculate the `duration` for a note at a given tempo, divide the beats per minute by 60 to get the number of beats per second. Then divide the beats per second into 2000, which is the number of half-milliseconds in a second. Multiply this ratio with the number of beats the note should receive. For example, in a 4/4 time signature each sixteenth note receives 1/4th of a beat. If an application is playing a song in 120 beats per minute and wanted four sixteenth notes, each `noteCmd` would have a `duration` of 250.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1					1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	**60**	**0**	61	62	63	64	65	66	67 68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

Table 4 noteCmd values

Table 4 shows the values that can be sent with a `noteCmd`. Middle C is represented by a value of 60. These values correspond to MIDI note values.

SUMMARY OF THE SOUND MANAGER

Sound Manager Constants

```
{ sound command numbers }
nullCmd           = 0;  {utility generally sent by Sound Manager}
initCmd           = 1;  {utility generally sent by Sound Manager}
freeCmd           = 2;  {utility generally sent by Sound Manager}
quietCmd          = 3;  {utility generally sent by Sound Manager}
flushCmd          = 4;  {utility generally sent by Sound Manager}
waitCmd           = 10; {sync control sent by application or modifier}
pauseCmd          = 11; {sync control sent by application or modifier}
resumeCmd         = 12; {sync control sent by application or modifier}
callBackCmd       = 13; {sync control sent by application or modifier}
syncCmd           = 14; {sync control sent by application or modifier}
emptyCmd          = 15; {sync control sent by application or modifier}
tickleCmd         = 20; {utility sent by Sound Manager or modifier}
requestNextCmd    = 21; {utility sent by Sound Manager or modifier}
howOftenCmd       = 22; {utility sent by Sound Manager or modifier}
wakeUpCmd         = 23; {utility sent by Sound Manager or modifier}
availableCmd      = 24; {utility sent by application}
versionCmd        = 25; {utility sent by application}
noteCmd           = 40; {basic command supported by all synthesizers}
restCmd           = 41; {basic command supported by all synthesizers}
freqCmd           = 42; {basic command supported by all synthesizers}
ampCmd            = 43; {basic command supported by all synthesizers}
timbreCmd         = 44; {noteSynth only}
waveTableCmd      = 60; {waveTableSynth only}
phaseCmd          = 61; {waveTableSynth only}
soundCmd          = 80; {sampledSynth only}
bufferCmd         = 81; {sampledSynth only}
rateCmd           = 82; {sampledSynth only}
continueCmd       = 83; {sampledSynth only}

{ synthesizer resource IDs used with _SndNewChannel }
noteSynth         = 1;        {note synthesizer}
waveTableSynth    = 3;        {wave table synthesizer}
sampledSynth      = 5;        {sampled sound synthesizer}
midiSynthIn       = 7;        {MIDI synthesizer in}
midiSynthOut      = 9;        {MIDI synthesizer out}

{ init options used with _SndNewChannel }
initChanLeft      = $02;      {left channel - sampleSynth only}
initChanRight     = $03;      {right channel- sampleSynth only}
initChan0         = $04;      {channel 0 - wave table only}
initChan1         = $05;      {channel 1 - wave table only}
initChan2         = $06;      {channel 2 - wave table only}
initChan3         = $07;      {channel 3 - wave table only}
initSRate22k      = $20;      {22k sampling rate - sampleSynth only}
initSRate44k      = $30;      {44k sampling rate - sampleSynth only}
initMono          = $80;      {monophonic channel - sampleSynth only}
initStereo        = $C0;      {stereo channel - sampleSynth only}
```

```
stdQLength        = 128;      {channel length for holding 128 commands}

{ sample encoding options }
stdSH             = $00       {standard sound header}
extSH             = $01       {extended sound header}
cmpSH             = $02       {compressed sound header}

{ Sound Manager error codes }
noErr             =    0      {no error}
noHardware        = -200      {no hardware to support synthesizer}
notEnoughHardware = -201      {no more channels to support synthesizer}
queueFull         = -203      {no room left in the channel}
resProblem        = -204      {problem loading the resource}
badChannel        = -205      {invalid channel}
badFormat         = -206      {handle to snd resource was invalide}
```

Sound Manager Data Types

```
Time        = LONGINT;

SndCommand  = PACKED RECORD
                cmd:            INTEGER; {command number}
                param1:         INTEGER; {first parameter}
                param2:         LONGINT; {second parameter}
              END;

ModifierStubPtr = ^ModifierStub;
ModifierStub = PACKED RECORD
                nextStub:       ModifierStubPtr; {pointer to next stub}
                code:           ProcPtr;    {pointer to modifier}
                userInfo:       LONGINT;    {free for modifier's use}
                count:          Time;       {used internally}
                every:          Time;       {used internally}
                flags:          SignedByte; {used internally}
                hState:         SignedByte; {used internally}
              END;


SndChannelPtr = ^SndChannel;
SndChannel = PACKED RECORD
                nextChan:       SndChannelPtr; {pointer to next channel}
                firstMod:       ModifierStubPtr; {ptr to first modifier}
                callBack:       ProcPtr;    {ptr to call back procedure}
                userInfo:       LONGINT;    {free for application's use}
                wait:           Time;       {used internally}
                cmdInProgress:  SndCommand; {used internally}
                flags:          INTEGER;    {used internally}
                qLength:        INTEGER;    {used internally}
                qHead:          INTEGER;    {used internally}
                qTail:          INTEGER;    {used internally}
                queue:          ARRAY [0..stdQLength-1] OF SndCommand;
              END;
```

```
SoundHeaderPtr = ^SoundHeader;
SoundHeader = PACKED RECORD             {sampled sound header}
                samplePtr:      Ptr;    {NIL if samples in sampleArea}
                length:         LONGINT; {number of samples in array}
                sampleRate:     Fixed;  {sampling rate}
                loopStart:      LONGINT; {loop point beginning}
                loopEnd:        LONGINT; {loop point ending}
                encode:         BYTE;   {sample's encoding option}
                baseNote:       BYTE;   {base note of sample}
                sampleArea:     PACKED ARRAY [0..0] OF Byte;
              END;


{refer to the Audio Interchange File Format "AIFF" specification}
ExtSoundHeaderPtr = ^ExtSoundHeader;
ExtSoundHeader = PACKED RECORD          {extended sample header}
                samplePtr:      Ptr;    {NIL if samples in sampleArea}
                length:         LONGINT; {number of sample frames}
                sampleRate:     Fixed;  {rate of original sample}
                loopStart:      LONGINT; {loop point beginning}
                loopEnd:        LONGINT; {loop point ending}
                encode:         BYTE;   {sample's encoding option}
                baseNote:       BYTE;   {base note of original sample}
                numChannels:    INTEGER; {number of chans used in sample}
                sampleSize:     INTEGER; {bits in each sample point}
                AIFFSampleRate:EXTENDED;{rate of original sample}
                MarkerChunk:    Ptr;    {pointer to a marker info}
                InstrumentChunks:Ptr;   {pointer to instrument info}
                AESRecording:   Ptr;    {pointer to audio info}
                FutureUse1:     LONGINT;
                FutureUse2:     LONGINT;
                FutureUse3:     LONGINT;
                FutureUse4:     LONGINT;
                sampleArea:     PACKED ARRAY [0..0] OF Byte;
              END;
```

Sound Manager Routines

```
FUNCTION SndDoCommand   (chan: SndChannelPtr; cmd: SndCommand;
                         noWait: BOOLEAN): OSErr;
      INLINE $A803;


FUNCTION SndDoImmediate (chan: SndChannelPtr; cmd: SndCommand): OSErr;
      INLINE $A804;


FUNCTION SndNewChannel  (VAR chan: SndChannelPtr; synth: INTEGER;
                         init: LONGINT; userRoutine: ProcPtr): OSErr;
      INLINE $A807;


FUNCTION SndDisposeChannel (chan: SndChannelPtr;
                            quietNow: BOOLEAN): OSErr;
      INLINE $A801;


FUNCTION SndPlay        (chan: SndChannelPtr; sndHdl: Handle;
                         async: BOOLEAN): OSErr;
      INLINE $A805;


FUNCTION SndControl     (id: INTEGER; VAR cmd: SndCommand): OSErr;
      INLINE $A806;


FUNCTION SndAddModifier (chan: SndChannelPtr; modifier: ProcPtr;
                         id: INTEGER; init: LONGINT): OSErr;
      INLINE $A802;


PROCEDURE MyCallBack    (chan: SndChannelPtr; cmd: SndCommand);


FUNCTION MyModifier     (chan: SndChannelPtr; VAR cmd: SndCommand;
                         mod: ModifierStub): BOOLEAN;
```

S

T, U, V, W, X, Y, Z